推酷

编程狂人

Programming Madman

NO.47

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:http://www.tuicool.com/mags/544e6201d91b145ab60124ec

欢迎下载推酷客户端体验更多阅读乐趣

目录

当当网开源Dubbox，扩展Dubbo服务框架支持REST风格远程调用

作者：沈理

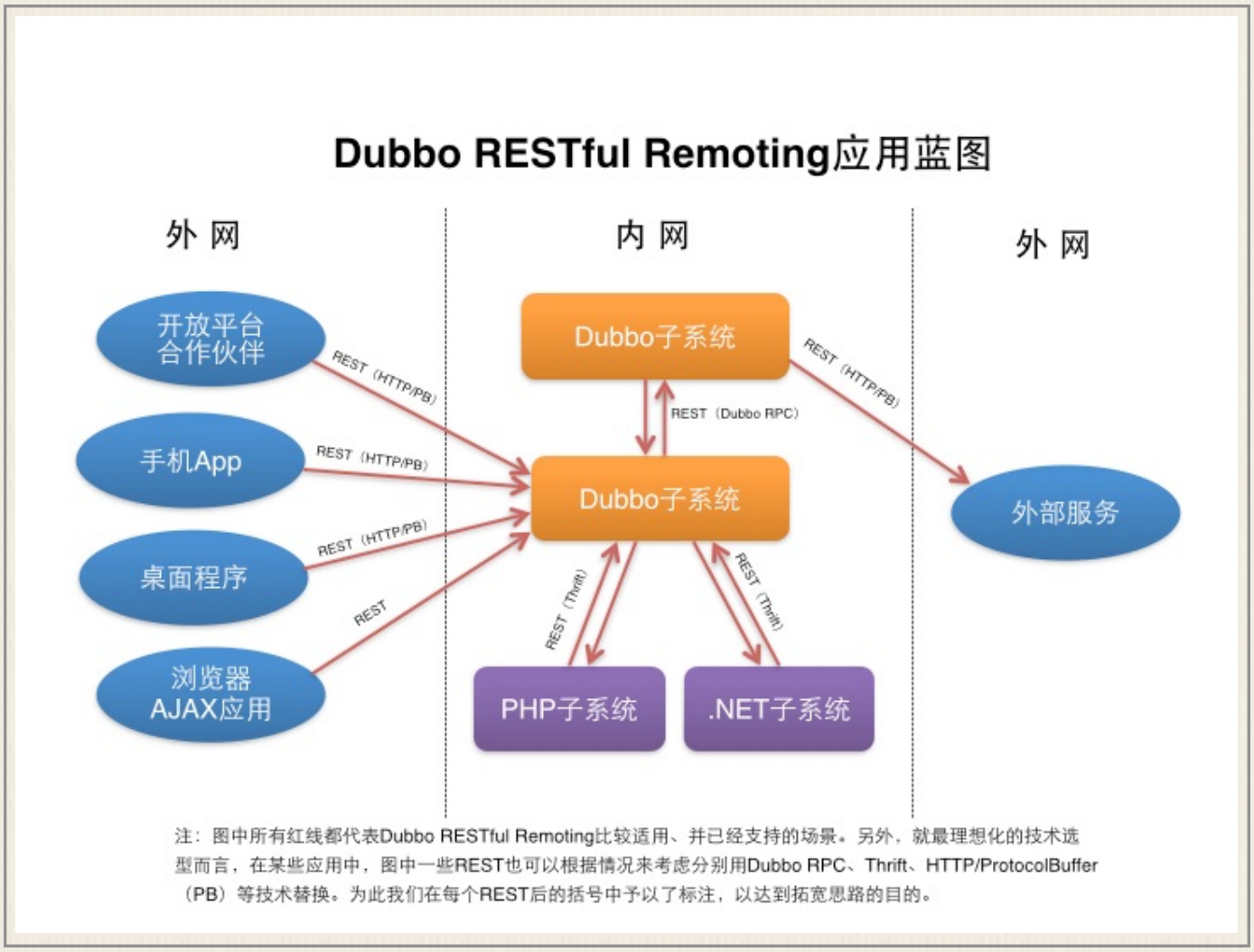
当当网近日开源了Dubbox项目，可为Dubbo服务框架提供多项扩展功能，包括REST风格远程调用、Kryo/FST序列化等等。

当当网架构部和技术委员会架构师沈理向InfoQ中文站介绍了Dubbox项目，开发背景和主要特点描述如下：

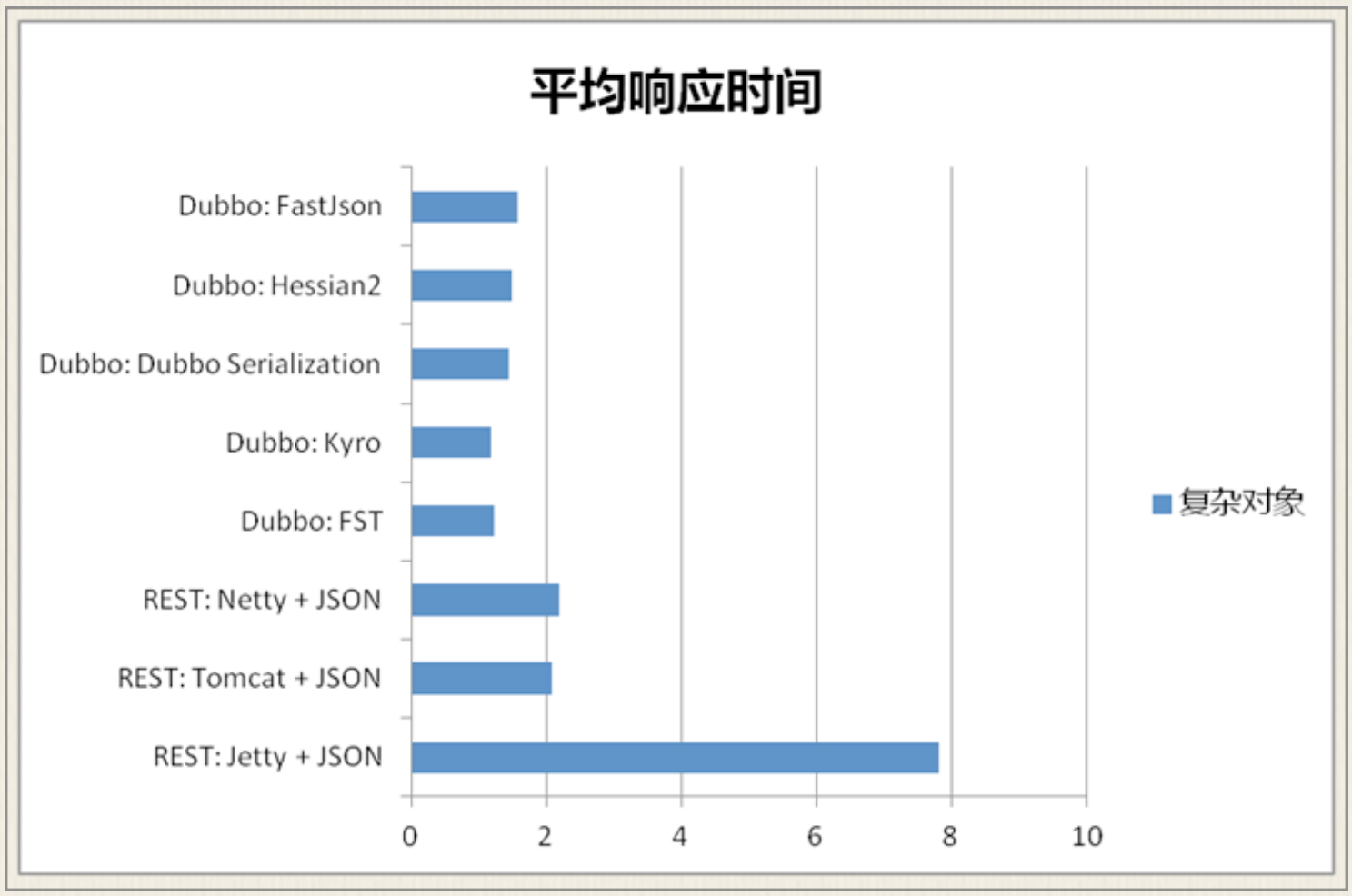
Dubbo是一个被国内很多互联网公司广泛使用的开源分布式服务框架，即使从国际视野来看应该也是一个非常全面的SOA基础框架。作为一个重要的技术研究课题，在当当网我们根据自身的需求，为Dubbo实现了一些新的功能，并将其命名为Dubbox（即Dubbo eXtensions）。

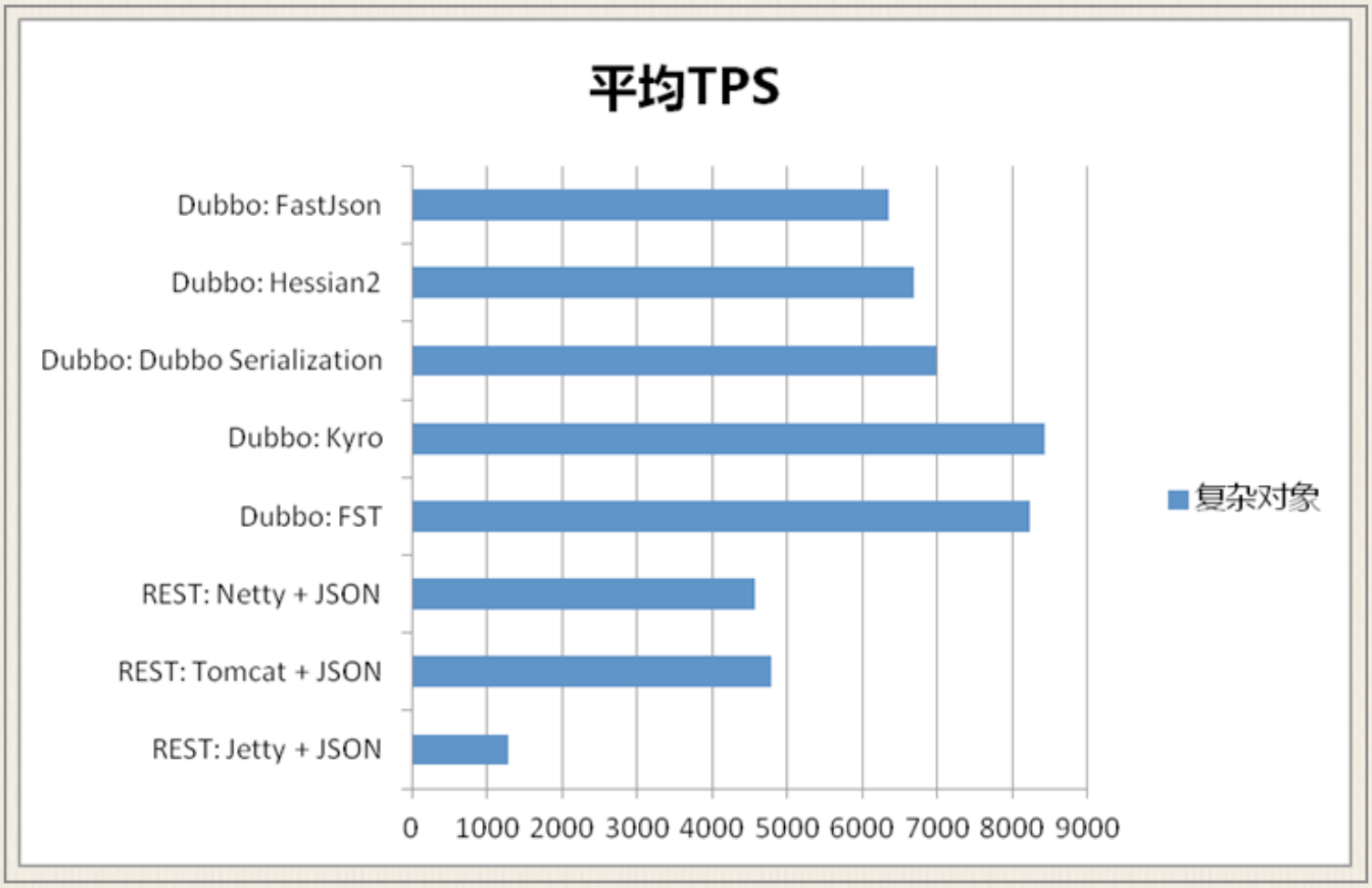
主要的新功能包括：

支持REST风格远程调用（HTTP + JSON/XML)：基于非常成熟的JBoss RestEasy框架，在dubbo中实现了REST风格（HTTP + JSON/XML ）的远程调用，以显著简化企业内部的跨语言交互，同时显著简化企业对外的Open API、无线API甚至AJAX服务端等等的开发。事实上，这个REST调用也使得Dubbo可以对当今特别流行的“微服务”架构提供基础性支持。 另外，REST调用也达到了比较高的性能，在基准测试下，HTTP + JSON与Dubbo 2.x默认的RPC协议（即TCP + Hessian2二进制序列化）之间只有1.5倍左右的差距，详见下文的基准测试报告。



支持基于Kryo和FST的Java高效序列化实现：基于当今比较知名的Kryo和FST高性能序列化库，为Dubbo 默认的RPC协议添加新的序列化实现，并优化调整了其序列化体系，比较显著的提高了Dubbo RPC的性能，详见下图和文档中的基准测试报告。





支持基于嵌入式Tomcat的HTTP remoting体系：基于嵌入式tomcat实现dubbo的 HTTP remoting体系（即dubbo-remoting-http），用以逐步取代Dubbo中旧版本的嵌入式Jetty，可以显著的提高REST等的远 程调用性能，并将Servlet API的支持从2.5升级到3.1。（注：除了REST，dubbo中的WebServices、Hessian、HTTP Invoker等协议都基于这个HTTP remoting体系）。

升级Spring：将dubbo中Spring由2.x升级到目前最常用的3.x版本，减少项目中版本冲突带来的麻烦。

升级ZooKeeper客户端：将dubbo中的zookeeper客户端升级到最新的版本，以修正老版本中包含的bug。

上面很多功能已在当当网内部稳定的使用，现在开源出来，供大家参考和指正。也希望感兴趣的朋友也来为Dubbo贡献更多的改进。

注：dubbox和dubbo 2.x是兼容的，没有改变dubbo的任何已有的功能和配置方式（除了升级了Spring之类的版本）。另外，dubbox也严格遵循了Apache 2.0许可证的要求。

附：分布式服务框架与RPC框架

目前开源领域能找到的分布式服务框架也有不少，比较有代表性的包括Twitter的Finagle（基于Scala语言），Flipkart（印度最大的B2C网站）的Phantom（文档较少），Apache的Tuscany（有点陈旧，而且不是很适合互联网公司）等等，其实国内也有少数公司提供了开源Java服务框架，但dubbo在其功能完善性、架构优雅性、使用简便性等方面依然有其相对独特的优势，尽管dubbo绝大部分的开发都是2012年以前完成的。

另外，业界的开源RPC框架更是数量众多，难以计数，但是，一般的RPC框架和我们讨论的分布式服务框架还是具有相当大的距离，比如在远程调用的多协议、多序列化支持，完善的服务治理等等方面都有较多的缺失。也正是由于这种缺失，著名的Apache Thrift等框架在这里也可归为RPC框架，而主要构建在Thrift之上的Finagle、Phantom等框架更接近于完整的分布式服务框架（当然，Dubbo其实也支持Thrift，只是还不太完善）。

有关沈理

沈理，目前在当当网的架构部和技术委员会担任架构师，主要负责当当网的SOA实施（即服务化）以及分布式服务框架的开发。以前也有在BEA、Oracle、Redhat等外企的长期工作经历，从事过多个不同SOA相关框架和容器的开发。

原文链接：http://www.infoq.com/cn/news/2014/10/dubbox-open-source

Yahoo前端优化性能规则

作者：gecko34

只有10%～20%的最终用户响应时间花在了下载HTML文档上，其余的80%～90%时间花在了下载页面中的所有组件上。

——Steve Souders

规则1——减少HTTP请求（Minimize HTTP Requests）

只有10%～20%的最终用户响应时间花在接收请求的HTML文档上，剩下的80%～90%时间都花在HTML文档所引用的所有组件（图片、脚本、 样式表、Flash等）进行的HTTP请求上。因此，改善响应时间最简单的办法就是减少组件数量并由此减少HTTP请求数。减少组件数量通常会和产品设计的初衷相矛盾，因此，此处给出了一些技术：

图片地图（Image Maps）联合多个图片到一个单独的图片中。下载图片大小的总和保持不变，但是，通过减少HTTP请求数的方式加速了页面。图片地图适用于导航栏或其他超 链接中使用多个图片的情形。但是，在定义图片地图上的区域坐标时，如果采用手工方式很难完成且容易出错，而且除了矩形外几乎无法定义其他形状。

CSS Sprites使用CSS background-image和background-position属性将多个图片联合成一个独立的图片来显示。它通过合并图片减少了HTTP请求，并且比图片地图更加灵活，同时也降低了图片的下载量。如果在页面中需要为背景、按钮、导航栏、链接等提供大量图片，CSS Sprites是一种优秀的解决方案。

内联图片（Inline images）使用data: URL scheme模式将图片嵌入到HTML文档中。通过此模式嵌入图片，不需要任何额外的HTTP请求开销。但是，目前的主流浏览器（主要是IE）不支持此种方式。

合并文件（Combined files）通过将所有JavaScript脚本合并到一个文件，所有CSS样式表合并到另一个文件的方式来减少HTTP请求的数量。但是简单的合并通常 会遇到模块化、页面变化等问题，需要根据页面引用脚本和样式表来具体分析以确定具体的组合方式。

规则2——使用内容发布网络（Use a Content Delivery Network）

用户同Web服务器的距离会对页面响应时间产生影响。网站最初通常将其所有服务器放在同一个地方，当用户群增加时，公司必须面对服务器放置地点不再合适的事实。因此，有必要在多个地理位置不同的服务器上部署内容。

作为实现地理位置分离的第一步，不应当首先尝试使用分布式架构重新设计Web应用程序。这样的应用程序决定了重新设计将带来如同步会话状态、在服务器放置地点间复制数据库事务等复杂问题。重新设计会推迟甚至根本无法实现缩短用户和网站内容距离的愿望。

如果应用程序Web服务器里用户更近，则一个HTTP请求的响应时间将被缩短；如果组件Web服务器离用户更近，则多个HTTP请求的响应时间将缩短。因此，与其重新开始设计应用程序，以便将应用程序Web服务器分散开，不如首先将组件Web服务器分散开。这不仅能达到响应时间大幅减少的目的，还很 容易实现。

内容发布网络（CDN）是一组分布在多个不同地理位置的Web服务器，用于更加有效的向用户发布内容。向特定用户发布内容的服务器基于对网络可用度的测量，例如，CDN可能选择网络阶跃数最小的服务器，或者具有最短响应时间的服务器。

除了缩短响应时间外，CDN还可以带来其他优势，包括备份、扩展存储能力和进行缓存；同时，CDN还有助于缓和Web流量峰值的压力，如在获取天气 或股市新闻、浏览体育或娱乐事件时。依赖CDN的一个缺点是网站

的响应时间会受到其他网站——甚至可能是竞争对手流量的影响；另一个缺点是无法直接控制组 件服务器所带来的问题。

CDN用于发布静态内容（如图片、脚本、样式表、Flash）。提供动态HTML页面会引入特殊的存储要求——数据库连接、状态管理、验证、硬件和OS优化等，这些复杂性超过了CDN的范围。另一方面，静态文件更容易存储并具有较少的依赖。

规则3——添加Expires头（Add an Expires or a Cache-Control Header）

Web页面包含大量组件，并且数量在不断增长。页面的初访者会进行很多的HTTP请求，但通过一个长久的Expires头，可以使这些组件被缓存下来，可以在后续的页面浏览中避免不必要的HTTP请求。长久的Expires头最长用于图片，但应该将其用于所有组件上，包括脚本、样式表和Flash。

Web服务器使用Expires头告诉Web客户端它可以使用一个组件的当前副本，知道指定时间为止。HTTP规范中简要的称该头为“在这一日期/时间之后，响应将被认为是无效的”。例如：

Expires: Thu, 15 Apr 2010 20:00:00 GMT

告诉浏览器该响应的有效性持续到2010年4月15日。

因为Expires头使用一个特定的时间，它要求服务端和客户端的时钟严格同步；另外，过期日期需要经常检查，一旦过期日期到了，需要在服务器中配 置提供一个新的日期。所以，HTTP1.1引入了Cache-Control头来克服Exipres头的限制。Cache-Control使用max- age指令指定组件被缓存多久，它以秒为单位定义了一个更新窗。使用带有max-age的Cache-Control可以消除Expires的限制，但对 于不支持HTTP1.1的应用，仍希望使用Expires头。可以同时制定这两个响应头，如果两者同时出现时，HTTP规范规定max-age指令将重写 Expires头。

当出现了Expires头时，直到过期时间为止一直会使用缓存的版本，浏览器不会检查任何更新，直到过了过期时间。为了确保用户能够获取组件的最

新版本，需要在所有的HTML页面中修改组件的文件名。Yahoo在此使用了将版本号嵌入在组件的文件名中的方法。

规则4——压缩组件（Gzip Components)

压缩组件通过减少HTTP请求产生的响应包的大小，从而降低传输时间的方式来提高性能。从HTTP1.1开始，Web客户端可以通过HTTP请求中的Accept-Encoding头来标识对压缩的支持：

Accept-Encoding: gzip, deflate

如果Web服务器看到请求中的这个头，就会使用客户端列出的方法中的一种来压缩响应。Web服务器通过响应中的Content-Encoding头来通知Web客户端：

Content-Encoding: gzip

目前许多网站通常会压缩HTML文档，脚本和样式表的压缩也是值得的（包括XML和JSON在内的任何文本响应理论上都值得被压缩）。但是，图片和PDF文件不应该被压缩，因为它们本来已经被压缩了。

压缩通常能将响应的数据量减少近70%，但是压缩通常情况下会带来服务端和客户端的CPU开销，要检测受益是否大于开销，需要综合考虑响应大小、连接的带宽和客户端也服务端直接的距离等因素。通常需要对大于1KB或2KB的文件进行压缩。

当浏览器通过代理来发送请求时，有可能出现浏览器期望接受的压缩后内容和实际接收到的不一致的情况。解决这一问题的方法是在Web服务器的响应中添 加Vary头。Web服务器可以告诉代理根据一个或多个请求头来改变缓存的响应。由于压缩的决定是基于Accept-Encoding请求头的，因此需要 在服务器的Vary响应头中包含Accept-Encoding:

Vary: Accept-Encoding

目前大约90%的通过浏览器进行的Internet通信都需要使用gzip，使得服务端和客户端的对等性变得额外重要。无论是客户端还是服务端发送 错误，都会造成页面被破坏。避免错误的一种方式是采用“浏览器白名单”方式，即只为经过证实支持压缩的浏览器提供压缩内容，但是当代理缓存加进

来以后，处 理边缘情形浏览器将变得更加复杂。另一种方式是使用Vary: *或Cache-Control: private头来禁用代理缓存。此种方式会为所有浏览器禁用代理缓存，从而增加带宽开销。如何平衡压缩和代理支持需要在加快响应时间、减小带宽开销和边 缘情形浏览器缺陷之间进行权衡：

如果网站的用户很少，并且他们处于一个小圈子中，边缘情形浏览器不需要太多关注，可以压缩内容并使用Vary: Accept-Encoding。

如果更注重带宽开销，可以和前一种情形一样，压缩内容并使用Vary: Accept-Encoding。

如果网站拥有大量的、多变的用户群，能够应付较高的带宽开销，并且享有高质量的名声，需要压缩内容并使用Cache-Control: Private。（Google和Yahoo都使用这种方式）

规则5——将样式表放在顶部（Put Stylesheets at the Top）

我们都希望页面能够逐步加载，也就是说，我们希望浏览器能够尽快显示内容。当浏览器逐步加载页面时，页头、导航栏、顶端logo等所有这些都会等待 页面的用户提供视觉反馈，这改善了用户体验。将样式表放在底部，为避免当样式变化时重绘页面中的元素，浏览器会阻塞内容逐步呈现。

样式表在页面中的位置并不影响下载时间，但是会影响页面的呈现。根据HTML规范“和A不一样，[LINK]只能出现在文档的HEAD节中，但其出现次数是任意的”。因此，问题的解决方式应该是遵循HTML规范，使用LINK标签将样式表放在文档的HEAD中。

规则6——将脚本放在底部（Put Scripts at the Bottom）

对响应时间影响最大的是页面中的组件数量，而脚本会阻塞组件的并行下载，带来性能上的问题。HTTP1.1规范建议浏览器从每个主机名并行下载两个 组件。如果一个Web页面平均将其组件分别放在两个主机名下，整体响应时间可以减少大约一半。我们可以通过对浏览器默认设置的修改来增加每个主机名并行下 载组件的数量，也可以使用CNAME（DNS别名）将组件

分别放到多个主机名下。但是，增加并行下载数量通常会带来性能上的开销，过多的并行下载有时反而 会降低性能。Yahoo!研究表明，使用两个主机名比使用1、4或10个主机名能带来更好的性能。

需要我们注意的是，下载脚本时并行下载实际上是被禁用的，即使此时使用了不同的主机名，浏览器也不会启动其他下载。因此，如果将脚本放在顶部，脚本 会阻塞后面内容的呈现，也会阻塞后面组件的下载。因此，放置脚本最好的地方是页面底部，这不会阻止页面内容呈现，而且页面中的可视组件可以尽早下载。

规则7——避免CSS表达式(Avoid CSS Expressions)

CSS表达式是动态设置CSS属性的一种强大（并且危险）的方式。它从IE5以后的版本被支持，在IE8中已经被废弃。
表达式的问题在于对其进行的求值频率比人们期望的要高。它们不只在呈现页面和大小改变时求值，当页面滚动，甚至用户鼠标在页面上移过时都要进行求值。

减少CSS表达式求值次数的一种方式是使用一次性表达式，如果CSS表达式必须被求值一次，可以在这一次中执行重写它本身。除此之外，还可以使用事件处理器来为特定的事件提供所期望的动态行为。

规则8——使用外部JavaScript和CSS（Make JavaScript and CSS External）

在现实环境中使用外部文件通常会产生较快的页面，因为JavaScript和CSS有机会被浏览器缓存起来。对于内联的情况，由于HTML文档通常 不会被配置为可以进行缓存的，所以每次请求HTML文档都要下载JavaScript和CSS。所以，如果JavaScript和CSS在外部文件中，浏 览器可以缓存它们，HTML文档的大小会被减少而不必增加HTTP请求数量。

决定是否使用外部文件的关键在于被缓存的外部文件占请求的HTML文档数的比重。如果网站用户在每次会话中进行多次页面访问，同时页面重用了多个脚本和样式表，使用外部文件时很好的选择。

对于大多数网站而言，难以精确度量以判断是否使用内联或外部文件，此时建议是使用外部文件的方式。对于这个问题的一个例外是网站主页，由于主页对于响应时间要求更高，因此更加倾向于内联而不是外部文件。

对于内联文件而言，由于无法利用浏览器缓存，因此给人感觉依然比较低效。我们可以通过加载后下载和动态内联的方式来使得网站主页既可以获得内联的优势，同时也能缓存外部文件。

规则9——减少DNS查找（Reduce DNS Lookups）

DNS对于网站来说会带来开销。通常浏览器查找一个给定主机名的IP要花费20～120毫秒的时间。在DNS查找完成之前，浏览器不能从此主机下载任何东西。

DNS查找可以被缓存起来以提高性能，这种缓存可以发生在ISP或局域网中的一台特殊的缓存服务器上，同时，缓存也会发生在独立的用户机器上。在用 户请求一个主机名后，DNS信息会留在操作系统的DNS缓存中，大多数浏览器也拥有自己的缓存，和操作系统缓存相分离。只要浏览器在其缓存中保留了DNS 记录，就不会通过操作系统来请求这个记录。

当客户端浏览器和操作系统中DNS缓存同时为空时，DNS查询的数量等于页面中唯一主机名的数量，这些主机名包括了页面的URL、图片、脚本、样式表、Flash等。所以，减少唯一主机名数量，可以减少DNS查询数。

减少唯一主机名数量会潜在的减少页面中并行下载的数量。避免DNS查找降低了响应时间，但减少并行下载可能会增加响应时间。对于这种情形，建议将这些组件放在至少2个，但不要超过4个主机名下。

规则10——精简JavaScript和CSS（Minify JavaScript and CSS）

精简是从代码中移除不必要的字符以减小其大小，进而改善加载时间。在代码被精简后，所有注释以及不必要的空白字符（空格、换行和制表符）都将被移除。对于JavaScript而言，因为需要下载的文件大小减小了，可以改善响应时间。

混淆是可以应用在源代码上的另一种优化方式。相比较于精简，混淆更加复杂，因此更容易产生bug。混淆可以更大程度上压缩源代码，但是也存在着一定的风险。

除了外部JavaScript外，内联在<script>和<style>块中的源代码也需要被精简。即使使用了gzip来压缩JavaScript和CSS，使用精简能够将代码大小再减少5%或者更多。

规则11——避免重定向（Avoid Redirects）

重定向用于将用户从一个URL路由到另一个URL。重定向有很多种，其中301和302是最常用的两种。下面是一个301响应头的示例：

HTTP/1.1 301 Moved Permanently

Location: http://example.com/newuri

Content-Type: text/html

浏览器会自动将用户带到Location字段给出的URL。重定向所必需的所有信息都包含在这个头中，响应体通常是空的。不管叫什么名字，301或 者302响应在实际中都不会被缓存，除非有附加的头（如Expires或Cache-Control等）要求它这么做。meta refresh标签和JavaScript也可以用于重定向，但是最好的技术是使用标准的3xx状态码，以保证后退按钮能够正常工作。

需要我们记住的是重定向会使页面变慢。在用户和HTML文档间插入一个重定向后，在此HTML文档到达之前页面上不会描绘任何东西，任何组件也不会被下载。

有一种重定向最为浪费，发生的也很频繁，但是Web开发人员通常都没有意识到它，它发生在URL的结尾必须出现斜线（/）而没有出现的情形。例如访问地址http://astrology.yahoo.com/astrology将导致一个301响应包含重定向至http://astrology.yahoo.com/astrology/。当主机名后缺少结尾斜线时不会发生重定向。在Apache中，我们可以通过Alias指令或者mod_rewrite模块或者DirectorySlash指令来处理缺少结尾斜线时的重定向问题。

从一个旧的站点链接到新的站点是使用重定向的另一种常见场景。其他形式还包括将一个网站的不同部分连接起来，以及基于一些条件（浏览器类型、用户帐 户类型等）来引导用户。使用重定向来连接两个网站很简单而且只需要很少的额外代码。但是，虽然重定向降低了开发的复杂性，也损害了用户体验，通常可以进行 其他的选择：如果两个代码的路径在同一台服务器上，可以使用Alias和mod_rewrite；如果域名由于重定向发生改变，可以使用一个 CNAME（一条DNS记录，用于创建一个域名指向另一个域名的别名）让两个主机名指向相同的服务器，然后使用Alias和mod_rewrite。

规则12——移除重复脚本（Remove Duplicate Scripts）

在一个页面中两次保护同一个JavaScript文件会损伤性能。导致一个脚本重复的因素主要有两个——团队大小和脚本数量。

当重复脚本的现象发生时，将产生不必要的HTTP请求和浪费执行JavaScript的时间。不必要的HTTP请求会发生在IE中，而不会发生在Firefox中。在IE中，如果脚本被包含两次且没有被缓存，浏览器会在页面加载期间产生两个HTTP请求；即使脚本可以缓存，当用户重新加载页面时也会产生额外的HTTP请求。对JavaScript进行的多余的执行从而浪费时间的现象在IE和Firefox中都存在，与脚本是否被缓存无关。

避免意外包含同一脚本两次的一种方法是在你的模块系统中实现一个脚本管理模块。包含脚本的典型方式是在HTML页面中使用SCRIPT标签：

<script type=”text/javascript” src=”menu_1.0.17.js”></script>

另一种选择是在PHP中创建一个函数：

为了防止统一个脚本被重复添加多次，insertScript函数需要添加处理脚本的依赖性和版本的功能。

规则13——配置Etag（Configure ETags）

实体标签（Entity Tag，ETag）是Web服务器和浏览器用于确认缓存组件的有效性的一种机制。ETag在HTTP1.1中引入，用于检测浏览器缓存中的

组件与原始服务 器上的组件是否匹配。ETag是唯一标识了一个组件的一个特定版本字符串。唯一的约束是该字符串必须用引号引起来。原始服务器使用Etag响应头来指定组 件的ETag。

HTTP/1.1 200 OK

Last-Modified: Tue, 12 Dec 2006 03:03:59 GMT

ETag: “10c24bc-4ab-457e1c1f”

Content-Length: 12195

此后，如果浏览器必须验证一个组件，它会使用If-None-Match头将ETag传回原始服务器。如果ETag是匹配的，就会返回304状态码，在此例中使响应减少12195字节。

GET /i/yahoo.gif HTTP/1.1

Host: us.yimg.com

f-Modified-Since: Tue, 12 Dec 2006 03:03:59 GMT

If-None-Match: “10c24bc-4ab-457e1c1f”

HTTP/1.1 304 Not Modified

ETag的问题在于，通常使用组件的某些属性来构造它，这些属性对于特定的、寄宿了网站的服务器来说是唯一的。当浏览器从一台服务器上获取了原始组 件之后又尝试向另一台服务器来验证组件时，ETag是不匹配的。这种情况对于使用服务器集群来处理请求的网站来说是很常见的一种情况。默认情况 下，Apache和IIS向E-
Tag中嵌入的数据都会大大降低有效性验证的成功率。

Apache1.3和2.x的ETag格式是inode-size-timestamp。文件系统使用inode来存储诸如文件类型、所有者、组和 访问模式等信息。尽管在多台服务器上一个给定的文件可能位于相同的目录、具有相同的文件大小、权限、时间戳等，从一台服务器到另一台服务器，器inode 仍然是不同的。
IIS5.0和6.0在ETag上有着类似的问题。IIS上ETag的格式是Filetimestamp:ChangeNumber。 ChangeNumber适用于跟踪IIS配置变化的计数器。对于一个网站背后的所有IIS服务器来说，ChangeNumber不大

可能相同。最终的结果是，对于完全相同的组件，从一台服务器到另一台，Apache和IIS产生的ETag是不会匹配的。如果ETag不匹配，用户就不会按照 ETag的设计计划那样接收到更小更快的304响应；相反，它们会收到普通的200响应以及组件的所有数据。如果只在一台服务器上部署网站，这通常不会产 生问题；但如果使用了服务器集群，同时使用Apache或者IIS进行默认的ETag配置，用户响应将变慢，服务器负载将变高，将消耗更多的带宽，同时代 理缓存的效率也会下降。即使组件具有长久的Expires头，一旦用户单击了Reload或Refresh按钮，依然会产生条件GET请求。
如果组件必须通过最新修改日期之外的一些东西来进行验证，则ETag是一种强大的方法；如果无须自定义ETag，则最好将其移除。Last- Modified头基于组件的时间戳进行验证，可以提供完全等价的信息，而且移除ETag可以减少响应和后续请求的HTTP头的大小。在Apache中， 只要向A-pache配置文件中简单地添加下面一行配置就能移除ETag：
FileETag none

规则14——使Ajax可缓存（Make Ajax Cacheable）

Ajax的一个最重要的优点就是向用户提供即时反馈，因为它异步的从后台Web服务器请求信息。但是，使用Ajax并不保证用户不会等到异步的JavaScript和XML返回响应。在很多应用程序中，用户是否需要等待取决于如何使用Ajax。用户是否需要等待的关键因素在于Ajax请求是主动 的还是被动的。主动请求是基于用户的当前操作而发起的，被动请求则是为了将来使用而预先发起的。我们需要注意的是，“异步”并没有暗示“实时”。

为了提升性能，最重要的是优化Ajax响应。而改善这些主动Ajax请求的最重要的方式就是使响应可缓存。如同在“添加Expires头”中讨论的，一些其他规则也适用于Ajax，包括：压缩组件、减少DNS查找、精简JavaScript、避免重定向、配置Etag。

原文链接：http://segmentfault.com/blog/gecko34/1190000000735395

参考链接：https://developer.yahoo.com/performance/rules.html

Web前端框架与类库的思考

作者：Ico_Coco

说起前端框架，我也是醉了。现在去面试或者和同行聊天，动不动就这个框架碉堡了，那个框架好犀利。

当然不是贬低框架，只是有一种杀鸡焉用牛刀的感觉。网站技术是为业务而存在 的，除此毫无意义，框架也是一样。在技术选型和架构设计当中，脱离网站业务发展的实际，一味的追求时髦新技术，可能会适得其反，将网站发展引入崎岖小道。 就好像一个日均pv只有几百的小型电商网站，却要大喊“某宝就是这么搞的”，然后搭建应用服务器集群，使用分布式文件系统和分布式数据库系统...等巴拉 巴拉的一堆用来处理高并发，海量数据访问的手段。我想说，有意义吗?

前端框架的理解误区

网站的价值在于它能为用户提供什么价值，在于网站能做什么，而不在于它是怎么 做的，所以在网站还很小的时候就去追求网站的架构框架是舍本逐末，得不偿失的。前端框架同理，如果是一个简单的页面型产品，应用只是依赖服务器来生成 Web页面和视图，并且只需要使用一些简单的Javascript或者JQuery来使应用更加具有互动性，那么一个JQuery前端类库就可以了，真的 没必要用上一些高大上的框架。

当然，框架的确是很有用的，重点是我们要知道什么时候该用什么框架。大公司大项目的经验和成功模式固然重要，值得学习借鉴，但我们不能因此变得盲从。只有深刻去理解前端框架，知道什么时候该用什么什么框架解决什么问题，才能有的放矢，直击要害。

前端框架与前端类库的区别

使用框架前，我觉得很重要的一点是弄清类库（诸如JQuery）和框架（诸如angularJS)的区别在何处。

简单而言，类库，解决的是代码或者是模块级别的复用或者对复杂度的封装问题，例如将一个解决复杂问题的功能模块封装成一个函数，提供一个简单的接口。库它是一种工具，它提供了很多封装好的方法，用与不用取决于我们自身，即使用了也不会影响我们呢的代码结构。

而框架，更多的是对模式级别的复用和对程序组织的规范。这里的模式是指比如 MVC，为了实现M和V的解耦，把复杂的耦合关系由经常变化的业务代码转移到不经常变化的框架内部消化。是面向一个领域来提供一套解决方案，提高开发效 率，如果我们选择了使用某框架，就应该遵循该框架所规定的规则。

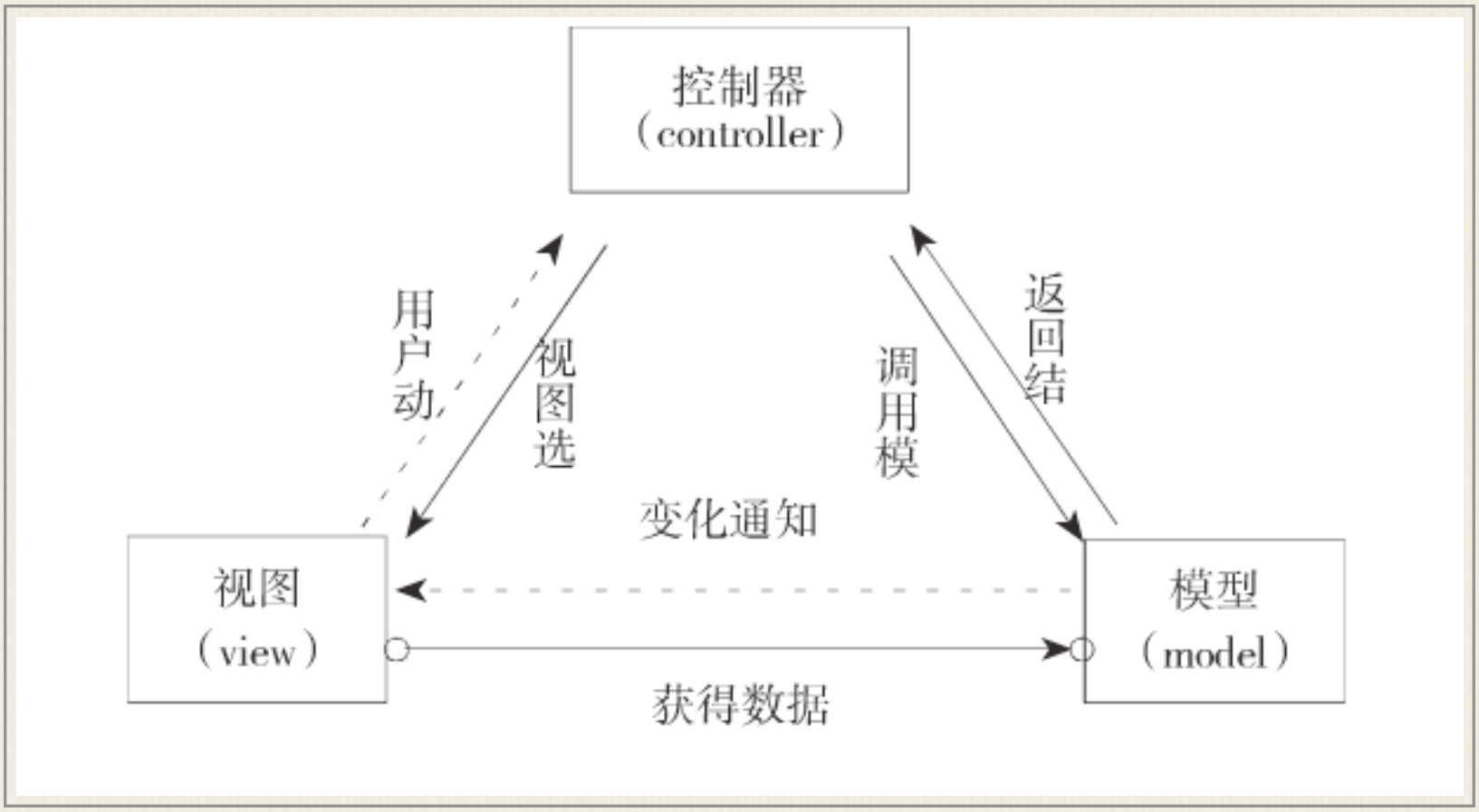
二者最主要的区别是：JQuery 以DOM操作为中心，框架，准确来说是 MVC框架，是以模型（model）为中心，而DOM操作是附加的。所以，以模型为中心最终达到的目的是带来一整套工作流程的变更，使得后台工程师可以编 写前端的模型代码，把后台与前端打通，交互设计师处理UI跟模型的互动关系，UI设计师可以专注、无障碍的处理HTML源码，把它们以界面模板的形式提交 给交互工程师。这一整套协作机制能大大提高开发效率。使用MVC框架使得前端任务更好的被解耦。

前端MVC框架思想

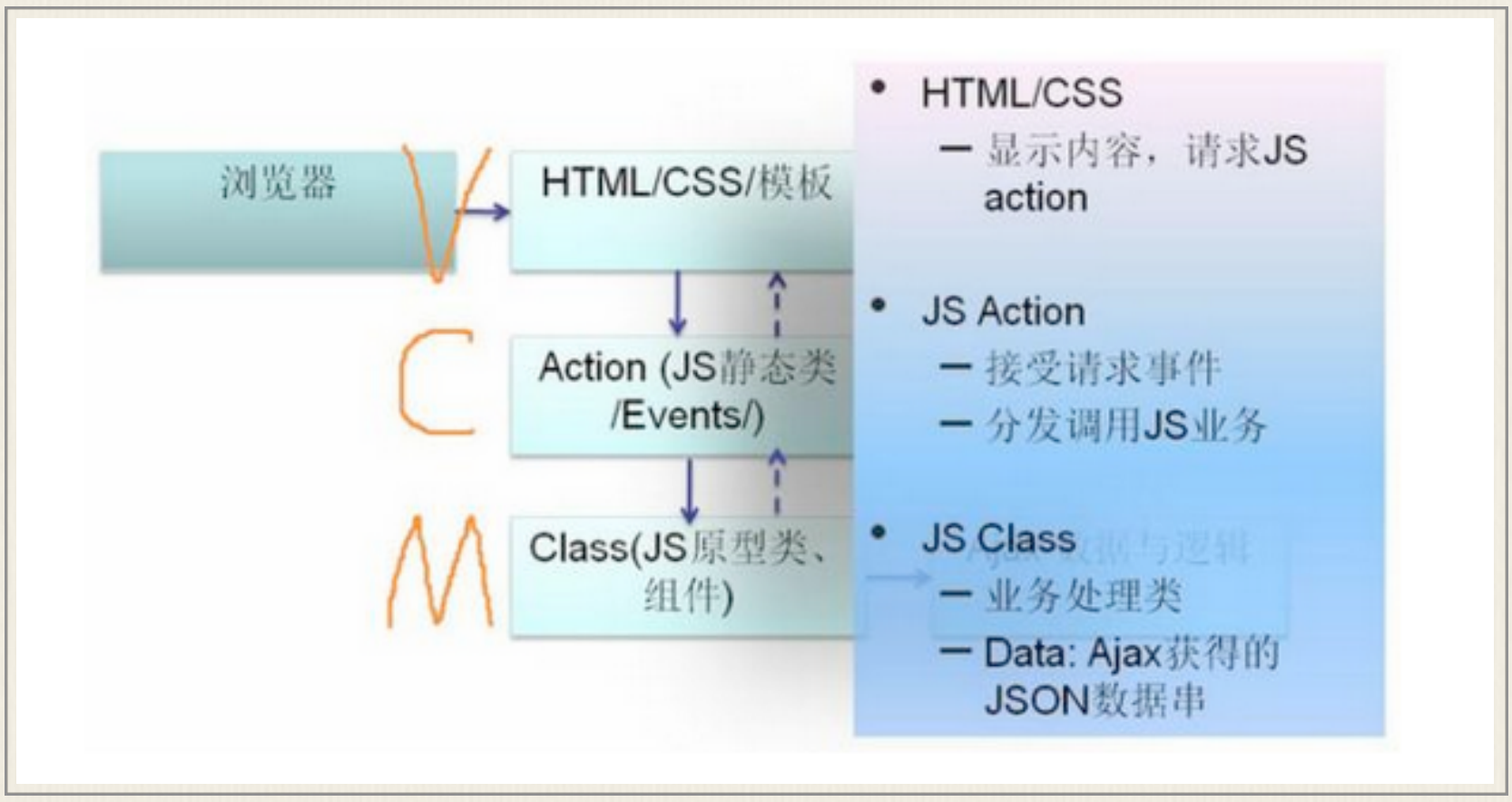
我们知道，传统的MVC模式将一个应用划分为——模型层（model）、视图层（view）、控制层（controller）。他们在应用系统中担当不同的角色，完成不同的任务。

- Model：即数据模型，用来包装和应用程序的业务逻辑相关的数据或者对数据进行处理，模型可以直接访问数据。
- View：视图用来有目的显示数据，在视图中一般没有程序上的逻辑，为了实现视图上的最新功能，视图需要访问它监视的数据模型。

- Controller： 控制器调控模型和视图的联系，它控制应用程序的流程，处 理事件并作出响应，事件不仅仅包括用户的行为还有数据模型上的改变。通过捕获用户事件，通知模型层作出相应的更新处理，同时将模型层的更新和改变通知给视 图，使得视图作出相应改变。因此控制器保证了视图和模型的一致性。



那么在前端中的表现。前端MVC中各部分的职责：



我对前端的View的理解是，与页面上元素直接相关的部分都属于View。包括html，CSS和一部分直接控制页面元素的JS。可以从Model中得到数据，并将其显示到页面上。而关于数据的变更与请求，则统统交给Controller处理。

那么Controller呢？作为Model和View的粘合 剂，Controller将View方面的请求转发给合适的Model，在必要时也会去更新View。而Controller本身也可以作为Model的 观察者，获取Model的变更。而作为Controller本身，就不应该有涉及到页面元素的代码了。

最后谈谈Model，与后端的沟通、A-JAX请求以及对数据的处理都属于 Model的工作。Model本身不知道谁是View，谁是Controller。它只提供一些方法供View和Controller调用，并且将变更通 知给它的观察者View或Controller。显然，Mod-el与页面元素之间也解耦了。

虽然基于 **MVC模型的框架之间也有很多不同之处，但是总体而言，Model负责保存vier需要的数据以及数据处理逻辑，例如读写，更新，删除，验证，转换等。 View负责接收并显示Model提供的数据以及接收用户的输入，并且响应事件，Model更新后及时将更新反馈回用户。Controller处理业务逻 辑和事件逻辑。**

知己知彼，对症下药

在前端框架和类库越来越丰富的今天。选择一款对的框架或类库就显得尤为重要了，我觉得没必要盲目跟风，看见什么框架火就屁颠屁颠跑去啃一个星期，然后因为项目工作上用不到，几个月之后又全忘光了。

所以我觉得重要的是把基础打扎实，重点是去了解各个类库与框架的作用，某类框架着重用于解决什么问题，然后在项目需要用到时候再去研读API才是上策。

最后，我们要清楚MVC在前端开发中的应用具有的局限性，简单的项目如果使用MVC框架会可能导致项目变得更加复杂。当然随着Web前端的复杂度不断增加，前端MVC框架的不断发展，相信在未来的应用软件类复杂产品当中，MVC框架一定会给前端工作带来效率上的飞跃。

以上只是我对前端框架和类库一些浅显的认识，不喜勿喷，更希望您能提出更好的学习框架和类库的方法。共同进步，共同学习。

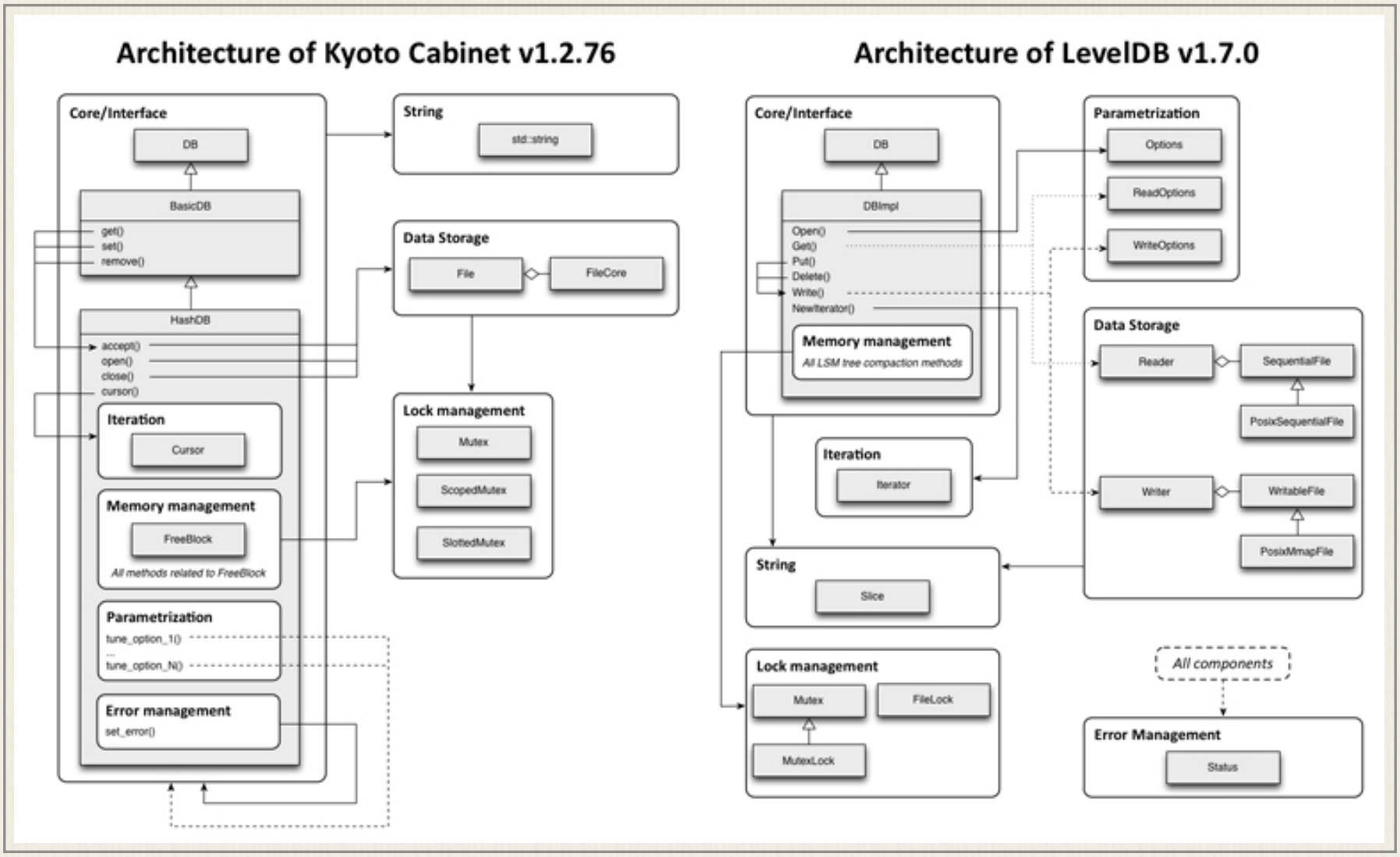
文中若有技术层面的错误还请斧正，误人子弟实乃罪过。

原文链接：http://www.cnblogs.com/coco1s/p/4040108.html

实现键值对存储（三）：Kyoto Cabinet 和LevelDB的架构比较分

作者：CuGBabyBeaR

在本文中，我将会逐组件地把Kyoto Cabinet 和 LevelDB的架构过一遍。目标和本系列第二部分讲的差不多，通过分析现有键值对存储的架构来思考我应该如何建立我自己键值对存储的架构。本文将包括：



1. 本架构分析的意图和方法

我曾经想过是应该写两篇独立的文章，一篇写LevelDB另一篇写Kyoto Cabinet，还是应该写一篇综合的文章。我相信软件架构是一门很需要决策的技艺，就如同建筑师需要考虑并选择每个部分的设计一样。方案不能孤立的评 估，而应该与其他方案之间进行权衡。软件系统架构的分析只能根据其背景在评价，并与其他架构比较。因此我将把键值对存储中遇到的主要组件过一遍，并比较现 有键值对系统的方案。我将会为Kyoto Cabinet 和 LevelDB使用我自己的分析，但其他项目我会使用现有的分析。这里是我选用的其他人的分析：

- BerkeleyDB, Chapter 4 in The Architecture of Open Source Applications, by Margo Seltzer and Keith Bostic (Seltzer being one of the two original authors of BerkeleyDB) [1]
- Memcached for dummies, by Tinou Bao [2]

- Memcached Internals [3]
- MongoDB Architecture, by Ricky Ho [4]
- Couchbase Architecture, by Ricky Ho [5]
- The Architecture of SQLite [6]
- Redis Documentation [7]

2. 键值对存储组件概述

尽管键值对存储的内部架构有很大不同，但总有相似的组件。下面列出了大部分键值对存储中遇到的主要组件及其功能的简述。

接口：键值对存储暴露给用户的一组方法和类，使用户可以与之互动。也叫做API。键值对存储的最小API包括Get(),、Put() 和Delete()方法。

参数系统：选项设置并传递给整个系统的其他组件。

数据存储：接口是用来访问内存中数据（也就是键和值）的。如果数据必须维护在持久性存储器中，例如硬盘或闪存，那么可能会出现同步性问题和并发性问题。

数据结构：用算法和方法来组织数据，并允许高效的存储的检索。通常使用哈希表或者B+树。LevelDB中则是日志结构合并树。数据结构的选择基于数据的内部结构和底层数据存储方案。

内存管理：系统中用来管理内存的算法和技术。内存相当重要，如果数据存储用错误的内存管理技术来访问，会极大地影响性能。

遍历：对数据库中所有键和值进行枚举和顺序访问的方法。解决方案大多是迭代器和游标。

字符串：数据结构是用来访问字符串的。把字符串单独拿出来说或许看起来有些过分详细了，但对于键值对存储来说，大量的时间都用来传递和处理字符串，STL的std::string可能不是最佳方案。

锁管理：所有关系到并发访问（带有信号灯和互斥的）内存区锁的机制，以及当数据存储是文件系统时的文件锁。同时处理关于多线程的问题。

错误管理：用来拦截和处理系统中遇到的错误的技术。

日志：记录系统中发生的事件的机制。

事务管理：能够确保所有操作正常执行的一系列操作的机制，并且在出现错误时，确保没有操作被执行且数据库也没有更改。

压缩：用来压缩数据的算法

比较器：用来比较两个键是否相同的方法。

校验和：用了测试并确保数据的完整性。

快照：快照提供其创建时全部数据库的只读镜像。

分区：也被称为分片，其包括将整套数据分配到多个数据存储中，可能是网络中的多个节点。

数据备份：为了防止系统或者硬件错误，确保持久性，一些键值对存储允许数据（或者数据分区）有数个同时维护的拷贝，最好是在多个节点上。

测试框架：用来测试系统的框架，包括单元测试和整体测试。

3. Kyoto Cabinet和LevelDB结构和概念的分析

下述关于LevelDB和Kyoto Cabinet的分析将集中在下列组件：参数系统、数据存储、字符串和错误管理。关于接口、数据结构、内存管理、日志和测试框架这些组件将包含在IKVS 系列之后的文章中。至于其他的组件，我目前不打算讲。其他系统，例如关系型数据库，有其他的诸如命令处理器、请求处理器、以及计划/优化器之类的组件，但 它们已经超出了IKVS系列的内容。

在我开始分析之前，请注意我认为Kyoto Cabinet 和 LevelDB是很出色的软件部分，我也很尊敬它们的作者。即便我说了关于他们的设计的坏话，要记得的是他们的代码仍然很出色，而我并没有像他们那样的才 华。这就是说，下边的文章是我对于Kyoto Cabinet 和 LevelDB代码的一点意见。

3.1 用Doxygen建立代码图

为了理解Kyoto Cabinet 和LevelDB的架构，我需要挖掘它们的代码。但是我也用Doxygen，一个用来浏览应用模块结构和类的非常强大的工具。Doxygen是一个适用于多个编程语言的文档系统，它可以直接从源代码中创

建报告文档或者HTML网站格式的文档。然而Doxygen同样可以用在没有 注释的代码中，并创建基于系统组织方式（文件、命名空间、类和方法）的接口。

你可以从官网上获得Doxygen [8]。在你机器上安装好Doxygen之后，只需要打开shell界面，到包含所有你需要分析的源代码的目录下。然后输入如下命令即可创建默认设置文件。

```
$
  doxygen -g
```

这将创建一个叫“Doxygen”的文件。打开这个文件，确认下述所有设置都设置为“yes”:EXTRACT_ALL, EXTRACT_PRIVATE, RECURSIVE, HAVE_DOT, CALL_GRAPH, CALLER_GRAPH。这些选项会保证从代码中抽取所有对象，包括子目录，并创建调用图。所有可用设置的描述可以在Doxygen的在线文档中找到[9]。只需要输入下面的命令即可用已选好的设置来创建文档。

```
$
  doxygen Doxygen
```

文档将在“html”文件夹中创建，你可以用任何web浏览器打开“index.html”文件来访问文档。你可以浏览代码，查看类之间的继承关系，并通过图来查看每个方法由其它哪个方法调用。

3.2 整体架构

图3.1和3.1分别是Kyoto Cabinet v1.2.76 和LevelDB 1.7.0的架构。类以UML类图标准表示。组件以圆角矩形表示，黑箭头表示其它实体调用了这个实体。从A到B的黑箭头表示A使用或者访问了B的元素。

这些图示表示的功能架构和结构架构基本相同。以图3.1为例，很多组件出现在HashDB类内部，因其这些组件的代码被定义为HashDB类的一部分。

依据内部组件的组织方式来比较，LevelDB是大赢家。原因是Kyoto Cabinet中，遍历、参数设置、内存管理和错误管理的组件都作为内核/接口组件的一部分，如图3.1所示。这使得这些组件和内核之间形成了强耦合，

并 局限了系统的模块化和功能扩展性。与之相反，LevelDB是以一种非常模块化的方法建立的，只有内存管理才是内核组件的一部分。

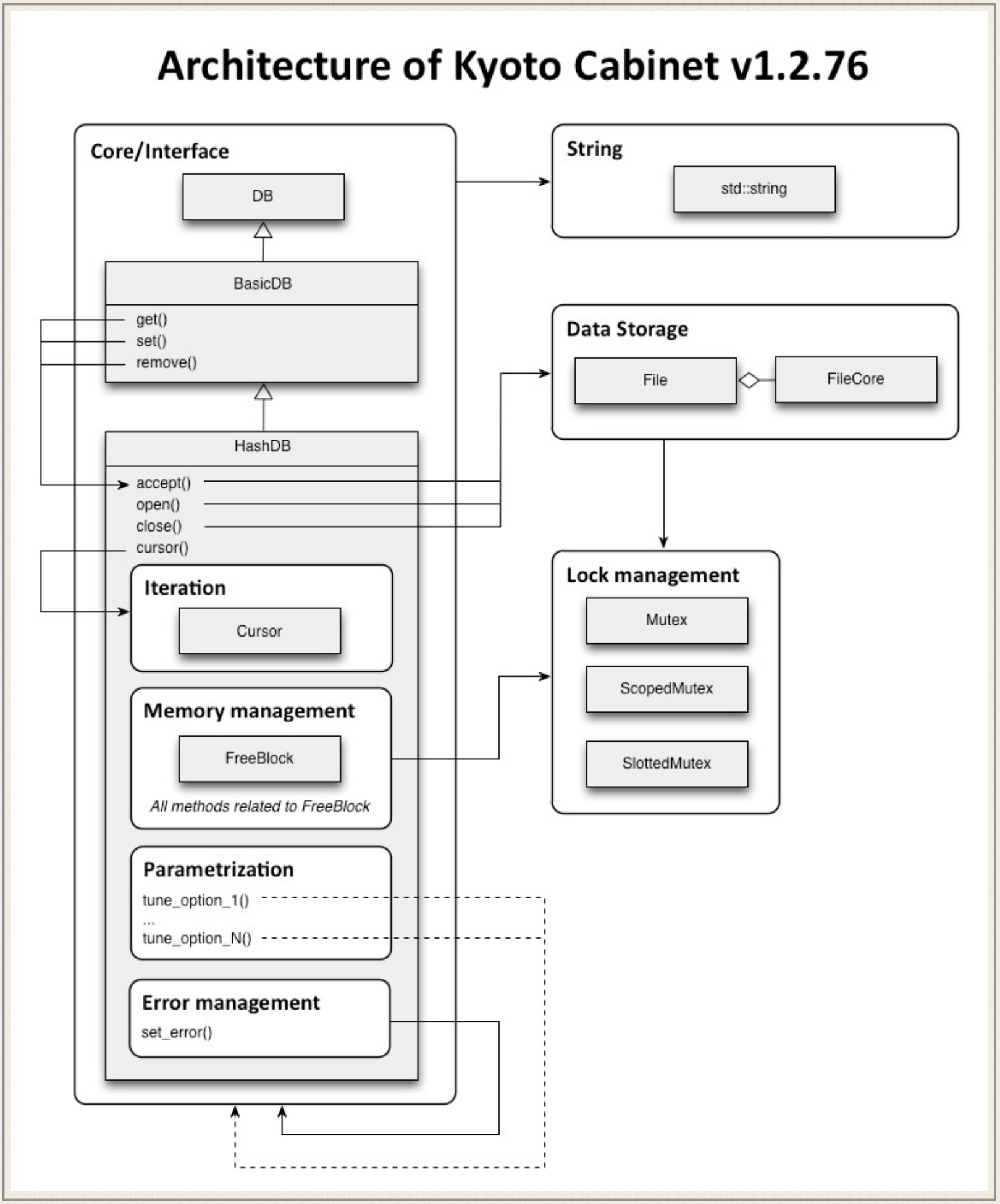


图3.1

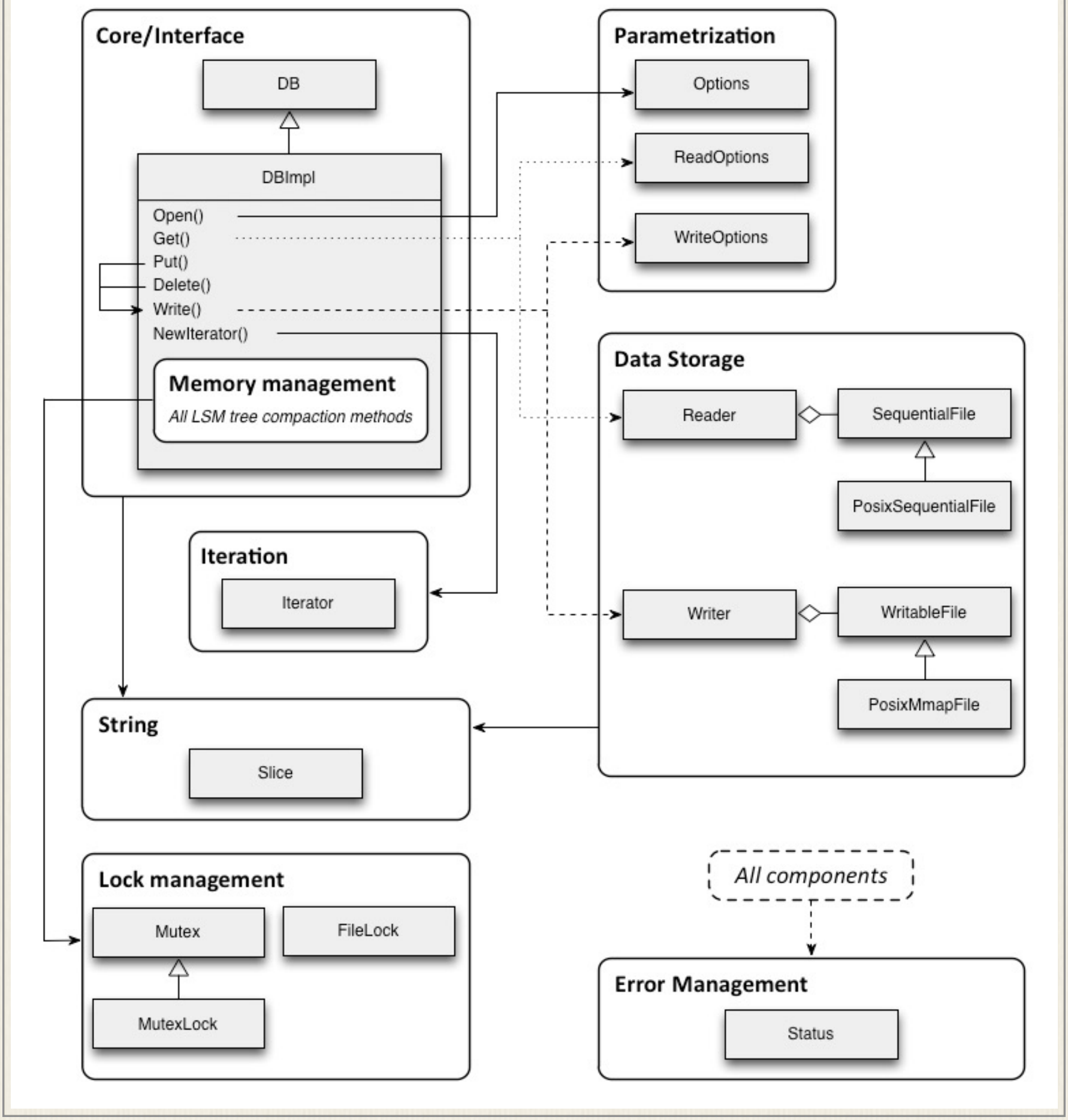

Architecture of LevelDB v1.7.0
Core/Interface
DB
DBImpl
Open()
Get()
Put()
Delete()
Write()
NewIterator()
Memory management
All LSM tree compaction methods
Parametrization
Options
ReadOptions
WriteOptions
Data Storage
Reader
SequentialFile
PosixSequentialFile
Writer
WritableFile
PosixMmapFile
Iteration
Iterator
String
Slice
Lock management
Mutex
FileLock
MutexLock
All components
Error Management
Status

图3.2

3.3 接口

Kyoto Cabinet 的HashDB类暴露出来至少50个方法，与之相比的是LevelDB的DBImpl类只有15个方法（其中4个还是测试用的）。这是Kyoto Cabinet的Core/Interface组件强耦合的直接结果。

API设计将会在将来的IKVS系列中详细讨论。

3.4 参数设置

在Kyoto Cabine中，参数是通过调用HashDB类的方法来调节的。有15个以“tune_”开头的方法来完成这个工作。

在LevelDB中，参数被定义在特定的对象中。“Options”对象中是通用参数，“ReadOptions”和“WriteOptions”中是 Get()和Put()分别需要的参数，如图3.2中所示。种子解耦提供了比较好的选项的扩展性，而不必像Kyoto Cabinet中调用Core中乱七八糟的公共接口。

3.5 字符串

在键值对存储中，随时都有大量的字符串处理。字符串被迭代、哈希、压缩、传递和返回。因此，巧妙的实现字符串类相当重要，每个对象节省一点，在大规模的运用上将会在全局造成引人注目的影响。

LevelDB使用一个特殊的类，称为“Slice” [10]。一个Slice包含一个字节数组以及数组的长度。这可以在O(1)的时间内获取字符串的长度，而不是std::string所需的O(n)而 不是对C的字符串调用strlen()时所需的O(n)。独立保存字符串长度也可以允许保存字符‘’，这表示键和值可以是真正的字节数组而非由null终 结的字符串。最后且最重要的是，Slice处理拷贝是通过创建一个浅拷贝，而非深拷贝。这表示它只简单地拷贝字节数组的指针，而不像std::string那样拷贝全部的字节数组。这避免了拷贝有可能出现的非常大的键或值。

像LevelDB一样，Redis使用他自己的数据结构来处理字符串。其目标同样是避免取字符串长度的时候避免使用O(n)操作[11]。

Kyoto Cabinet使用std::string作为字符串对象。

我的意见是，一个字符串类的实现适应于键值对存储的需求是非常必要的。如果能够避免，为什么要花费时间来拷贝字符串并分配内存呢?

3.6 错误管理

在我看过的键值对存储的所有C++源代码中，我没有见过一个将异常作为全局的错误管理系统使用。在Kyoto Cabinet中，kcthread.cc文件中的线程组件使用了异常，但我认为这个选择与其说是通用架构倒不如说是只是在处理线程而已。异常十分危险， 并应该尽可能的避免。

BerkeleyDB有很好的C风格的方法来处理错误。错误信息和代码集中在一个文件中。所有返回错误代码的函数都有一个叫“ret”的整型本地变量，这个变量将会在处理过程中赋值并在最后返回。这种方法贯穿在所有的文件和模块中：相当优雅和标准化的错误管理。在一些函数中使用了向前跳转的goto语句 ——一种在如Linux内核那样的纯C系统中广泛使用的技巧[12]。虽然这种方法十分简洁和干净，但C风格的错误管理方法不太适合C++应用。

Kyoto Cabinet中，错误对象存储在每个诸如HashDB的数据库对象中。在数据库类中，各个方法在出现错误的时候调用set_error()来设置错误对象，然后以很符合C风格的返回true或者false。不会像BerkeleyDB那样在方法末尾返回本地变量，返回语句出现在错误出现的地方。

LevelDB完全不使用异常，而是使用一个叫做Status的类。这个类有错误值和错误信息。每个方法都返回这个对象，这样错误状态既可以就地处理也可 以传递给调用栈中更高的其他方法。这个Status类错误码存储在字符串中，也是一种非常的聪明的实现。我对于这种设计方法的理解是，在大部分时间里，方 法将会返回一个“OK”的状态（Status）对象，以表示没有出现任何错误。这样，错误信息字符串是NULL，而这个Status对象的处理是相当轻量 的。如果Status对象增加一个属性来保存错误码，那么即便在“OK”状态的Status对象中仍需要给这个属性赋值，这即表示在每次调用方法的时候都 要用更多的空间。所有的组件都使用这个Status类，并且没必要像Kyoto Cabinet那样总要调用一个方法，如图 3.1 and 3.2所示。

错误管理的所有方案都在上文中讲过了，我个人比较推荐LevelDB使用的方案。这个方案避免使用了异常，也不是一个我看来相当局限的单纯的C风格的错误管理，并且其避免了像Kyoto Cabinet 那样与核心组件任何不必要的耦合。

3.7 内存管理

Kyoto Cabinet 和LevelDB都在内核组件中定义了内存管理。对于Kyoto Cabinet，内存管理一来可以跟踪数据库文件中临近的空块，二来当数据项保存的时候可以选择足够大小的块。而文件本身只是用mmap()函数映射出来 的内存空间。另外MongoDB也使用内存映射文件[13]。

而LevelDB使用的是一个日志结构合并树，其不像保存在硬盘上的哈希表那样文件中有未使用的空间。内存空间管理也包括一旦日志文件大小超过某值后，压缩这些文件的功能[14]。

其它如Redis之类的键值对存储，用malloc()来分配内存——在Redis的例子中，内存分配算法不是操作系统提供的dlmalloc或者ptmalloc3，而是jemalloc[15] 。

3.8 数据存储

Kyoto Cabinet, LevelDB, BerkeleyDB, MongoDB 和Redis使用文件系统来存储数据。与之相反Memcached 则是在内存中保存数据。

4. 代码审查

本节是对Kyoto Cabinet 和LevelDB的一个简单的代码审查。这个代码审查并不全面，并只包含了我在阅读源代码时觉得比较出色的元素。

4.1 声明和定义的组织

如果代码都像LevelDB那样正常的组织，声明都在.h头文件中，而定义都在.cc文件中。但我在Kyoto Cabinet中发现了一些令人震惊的事情。实际上，很多类中.cc文件并没有包含任何定义，而方法都直接在.h文件中定义。在其他文件中，一些方法 在.h中定义另一些在.cc文件中定义。虽然我理解这样做的背后可能有一些原因，但我仍认为在C++应用中不遵守这些惯例根本是错误的。之所以说是错的是 因为一来它让我像那样惊讶，二来我必须在两种不同的文件中找定义。

4.2 命名

首先，Kyoto Cabinet相对于Tokyo Cabinet.有了显著的改进。整体架构和命名规则都大幅改进了。尽管如此，我仍然发现Kyoto Cabinet中的很多名字都很晦涩，譬如属性和方法叫做embcomp、trhard、fmtver()、fpow()。这让人觉得C++代码中混进了 一些C代码。另一方面，LevelDB中的命名相当清晰，除了诸如mem、imm和in的一些临时变量。但这些不清晰的密码相当微量而代码可读性相当强。

4.3 代码重复

我在Kyoto Cabinet中确实看到了一些代码重复。这些用来文件碎片整理的代码至少重复了3次，而所有需要分为Unix和Windows两个版本的方法都显示出大 量的重复。我没有在LevelDB看到明显的代码重复，我相信应该也有一些，但需要挖掘的更深才能找到。这证明LevelDB的代码重复问题确实比 Kyoto Cabinet要小。

5. 参考文献

[1] http://www.aosabook.org/en/bdb.html

[2] http://work.tinou.com/2011/04/memcached-for-dummies.html

[3] http://code.google.com/p/memcached/wiki/NewUserInternals

[4] http://horicky.blogspot.com/2012/04/mongodb-architecture.html

[5] http://horicky.blogspot.com/2012/07/couchbase-architecture.html

[6] http://www.sqlite.org/arch.html

[7] http://redis.io/documentation

[8] http:://doxygen.org

[9] http://www.stack.nl/~dimitri/doxygen/config.html

[10] http://leveldb.googlecode.com/svn/trunk/doc/index.html

[11] http://redis.io/topics/internals-sds

[12] http://news.ycombinator.com/item?id=3883310

[13] http://www.briancarpio.com/2012/05/03/mongodb-memory-management/

[14] http://leveldb.googlecode.com/svn/trunk/doc/impl.html
[15] http://oldblog.antirez.com/post/everything-about-redis-24.html

原文链接：http://blog.csdn.net/cugbabybear/article/details/40405919

优化无极限：盘古Master优化实践

作者：吴均平

摘要：盘古是阿里云飞天平台的分布式文件系统。在飞天5K项目中，集群规模快速扩张到5000个节点，规模的迅速扩张，对盘古的性能带来了不小的挑战。本文分享了为确保高性能，盘古Master的优化方案。

盘古是一个分布式文件系统，在整个阿里巴巴云计算平台——“飞天”中，它是最早被开发出的服务，因此用中国古代神话中开天辟地的盘古为其命名，希冀能创建出一个全新的“云世界”。在“飞天”平台中，它是负责数据存储的基石性系统，其上承载了一系列的云服务（如图1所示）。盘古的设计目标是将 大量通用机器的存储资源聚合在一起，为用户提供大规模、高可用、高吞吐量和良好扩展性的存储服务。盘古的上层服务中，既有要求高吞吐量，期待I/O能力随 集群规模线性增长的“开放存储”；又有要求低时延的“弹性计算”，而作为底层平台核心模块的盘古必须二者兼顾，同时具备高吞吐量和低时延。

图1 飞天整体架构图

在内部架构上盘古采用Master/ChunkServer结构（如图2所示），Master管理元数据，多Master之间采用Primary- Secondari-es模式，基于PAXOS协议来保障服务高可用； ChunkServer负责实际数据读写，通过冗余副本提供数据安全；Client对外提供类POSIX的专有API，系统地提供丰富的文件形式，满足离 线场景对高吞吐量的要求，在线场景下对低延迟的要求，以及虚拟机等特殊场景下随机访问的要求。

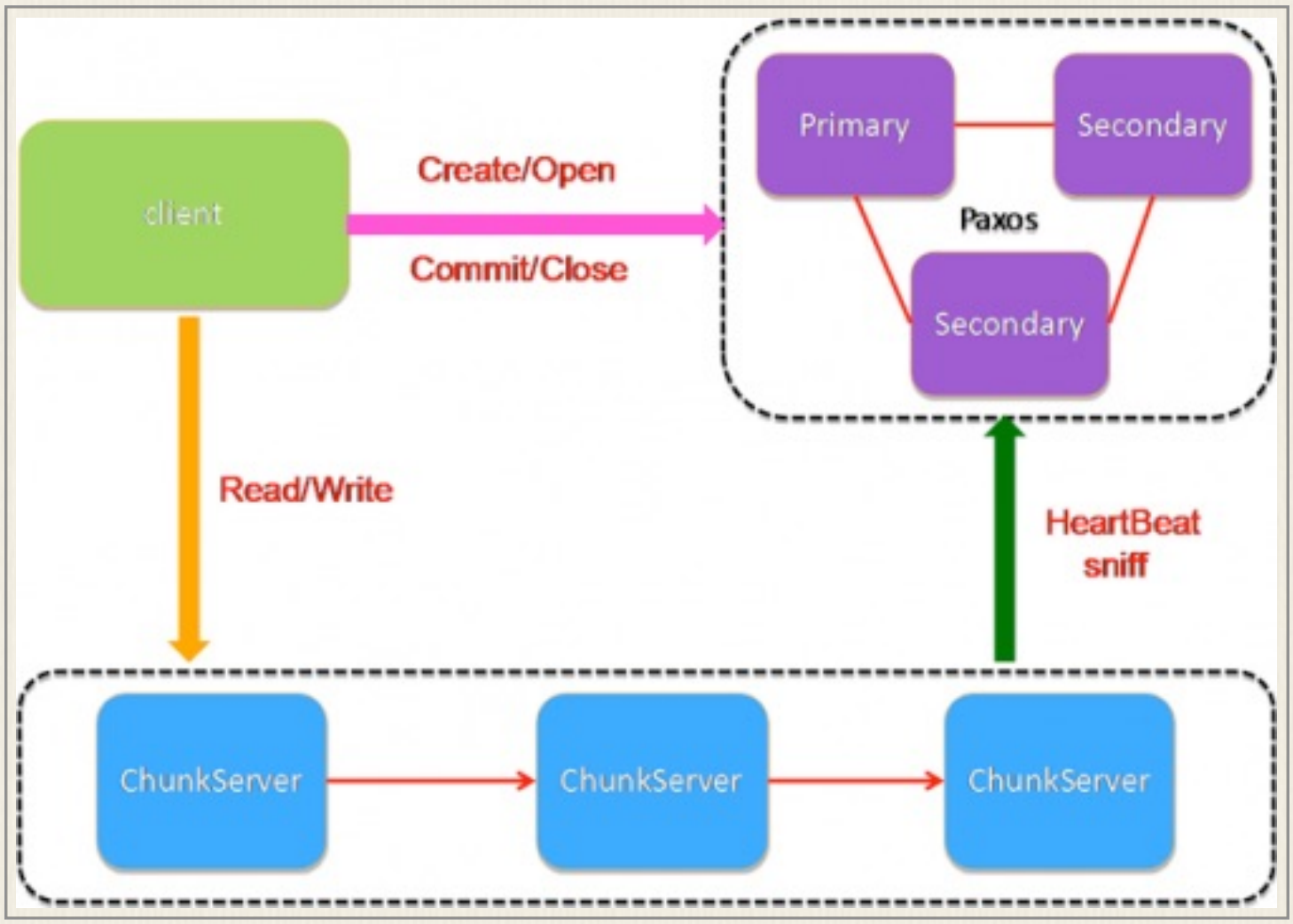


图2 盘古简明架构图

自5K项目以来，集群规模快速扩张到5000个节点，规模引发的相关问题纷至沓来。首当其冲的就是盘古Master IOPS问题，因为更大的集群意味着更多文件和更多访问，显然上层应用对存储亿级文件和十亿级文件集群的IOPS是有显著区别的。同时更大规模的集群让快 速发展的上层应用看到了更多可能性，导致更多业务上云，存储更多数据，也间接导致了对IOPS的更高需求。此前盘古Master较低的IOPS已经限制了 上层业务的快速扩张，业务高峰期时有告警，亟待提升。另外一个规模相关问题就是盘古Master

冷启动速度，更多的文件和Chunk数导致更长的冷启动时 间，影响集群可用性。

要解决上述问题，性能优化势在必行。但对于盘古这样复杂的大型系统，不可能毕其功于一役，需要在不同的阶段解决不同的 性能瓶颈。优化通常会伴随整个生命周期，因此优化工作本身也需要进行经验、工具等方面的积累，为后续持续优化提供方便。因此在这次规模问题优化中，我们积极建设了自己的锁Profile工具，并依此解决了多个锁导致的性能问题；在解决主要的锁瓶颈后，我们进行了架构上的优化，包括Pipeline优化和 Group Commit优化，取得了良好效果；最后我们通过对细节的不断深入，反复尝试，较好地解决了由规模导致冷启动时间过长的问题。

工具篇

在 盘古这样复杂的大型系统上，锁是优化过程经常遇到的问题。优化的首要任务是找出具体瓶颈，切忌猜测和盲目修改代码。先用压测工具对盘古Master加压， 通过Top、Vmstat 等系统工具发现CPU负载不到物理核的一半，Context Switch高达几十万，初步怀疑是锁竞争导致。同事也觉得某些路径下的锁还有优化空间，但到底是哪些锁竞争厉害？哪些锁持有的时间长？哪些锁等待时间 长？具体又是哪些操作需要这些锁？由于缺少切实可靠的数据支撑，不能盲目下手，而坚持数据说话是一个工程师应有的品质。

盘古Master提供了众多的读写接口，内部的不同模块使用了大量的锁，我们需要准确得知是哪类操作导致哪个锁竞争严重，此前常用的一些Profile工具难以满足需求，因此我们有必要打造自己的“手术刀”——锁分析工具，方便后续工作。

首先为了区分不同的锁，我们需要对代码中所有的锁进行统一命名，并将命名记录到锁实现内，同时为了区分不同的操作，还需要对所有的操作进行一个唯一的类型编 号。在某个Worker线程从RPC中读取到具体请求时，将类型编号写入到该Worker线程私有数据中，随后在该RPC请求的处理过程中，不同锁的拿锁 操作，都归类于该操作类型。

在每个锁内部，维护一个Vector数据， LockPerfRecord记录锁的Profile信息，具体定义如下：

```
struct  LockPerfRecord
{
    uint64_t  acquireCounts;
    uint64_t  waitingTime;
    uint64_t  occupyTime;
};
```

每个操作对应Vector中的一个元素，线程私有数据中记录的操作编号即Vector下标。整体来看形成了表1中的稀疏二维数组，空记录表示该操作未使用对应锁。

	Create	Open	Read	Write	Commit	ActionType X
Lock A	record				record	
Lock B			record		record	
Lock X	record				record	record

表1 操作类型与锁统计示意表

具体实现上，Acquire Counts采用原子变量，时间测量上采用Rdtsc。谈到Rdtsc有不少人色变，认为有CPU变频、多CPU之间不一致等问题，但在长时间粒度（分钟 级）和大量调用（亿次）的压测背景下，这些可能存在的影响不会干扰最终结果，多次实验结果也证实了这一点。其他实现细节，如定时定量（每n分钟或者每发起 x次操作）发起不影响主流程的Perf Dump就不再赘述。整个代码在百行左右，对平台的侵入也很小，仅需要为每个锁添加一个初始化命名，在RPC处理函数的入口设定操作编号即可。

利器在手，按图索骥，对存在性能问题的锁进行各种优化。例如使用多个锁来替换原来全局唯一锁，减少冲突概率；减少加锁范围；使用无锁的数据结构，或者使用更 轻量级的锁来优化。整个优化过程极富趣味性，经常是解决一个锁瓶颈后，发现瓶颈又转移到另外一个锁上，在工具的Profile结果中有非常直观的展示。先 后优化了Client Session、Placement等模块中锁的使用，取得了显著的效果，CPU基本可以跑满，Context Switch大幅度下降，整个过程酣战淋漓。

架构篇

在锁竞争问题解决到一定程度后，继续进行锁的优化就很难在IOPS上取得较大收益。此时结合业务逻辑，我们发现可以从架构上做出部分调整，以提升整体的IOPS。

读写分离（快慢分离）

整 个盘古Master对外接口众多，根据是否需要在Primary和Secondaries之间同步Operation Log来分成读和写两大类。所有读操作都不需要同步Operation Log，所有的写操作基于数据一致性必须同步。考虑到Primary和Secondaries随时都可能发生切换，要保证数据一致，Client端发送的 每一个写请求，Primary必须在同步Operation Log完全成功后才能返回Client。显而易见，写比读要慢得多，如果让同一个线程池来同时服务读和写，将会导致什么结果？一个形象的隐喻是多车道的高 速公路，如果一条高速公路上不按照速度来划分多车道，而是随心所欲地混跑，20码和120码的车跑在同一条车道上，整体的吞吐量无论如何也不会高。参照高 速公路设计，进行快慢分离，耗时低的读操作占用一个线程池，耗时高的写操作使用另外一个线程池，双方互不干扰，进行这样一个简单的切分后，读IOPS得到 显著提升，而写操作并未受影响。

Pipeline

读写分离后，读的IOPS性能得到极大提升，但写操作依然有待提升。写操作的基本流程如图3所示。

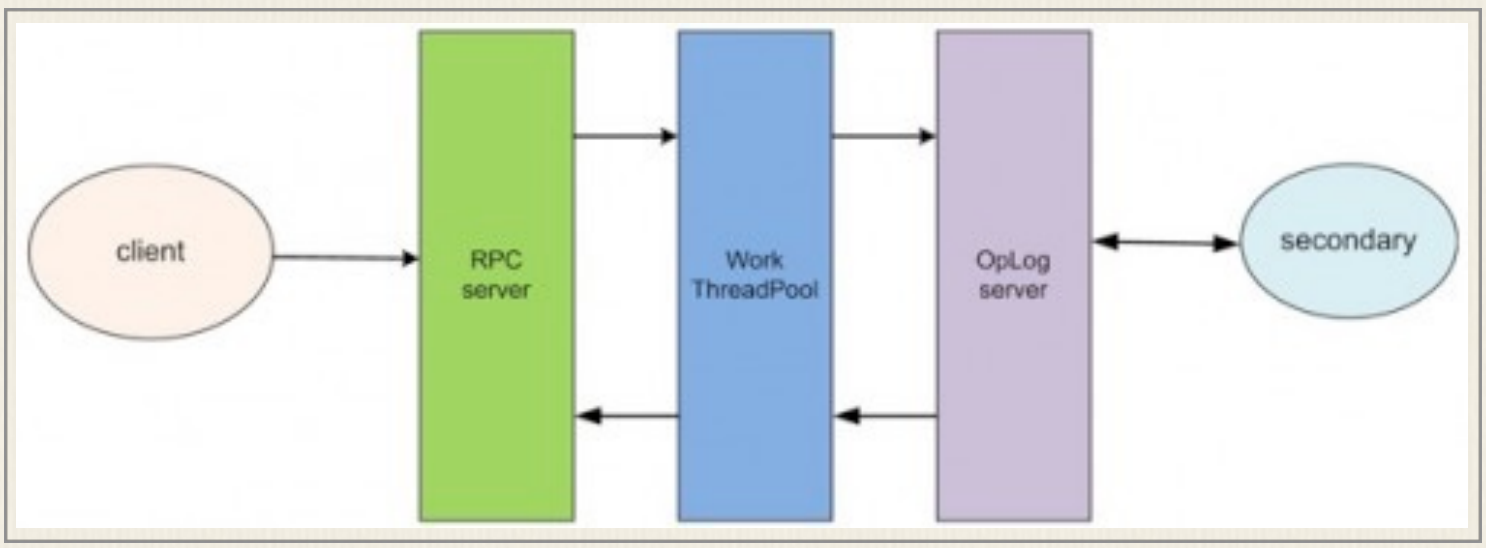


图3 写操作基本流程图

在Master端，写操作最大的时间开销在同步Operation Log到Secondaries。这里的关键缺陷在于同步Operation Log期间，工作线程只能被动地等待一段相当长的时间，这个过程包括将数据同步到Secondaries上，并由Secondaries将这个数据同步写 入到磁盘，由于涉及到同步写物理磁盘（而非写Page Cache），这个时间是毫秒量级。所以写操作的IOPS肯定不会高。定位问题后，在结构上稍做调整如图4所示。

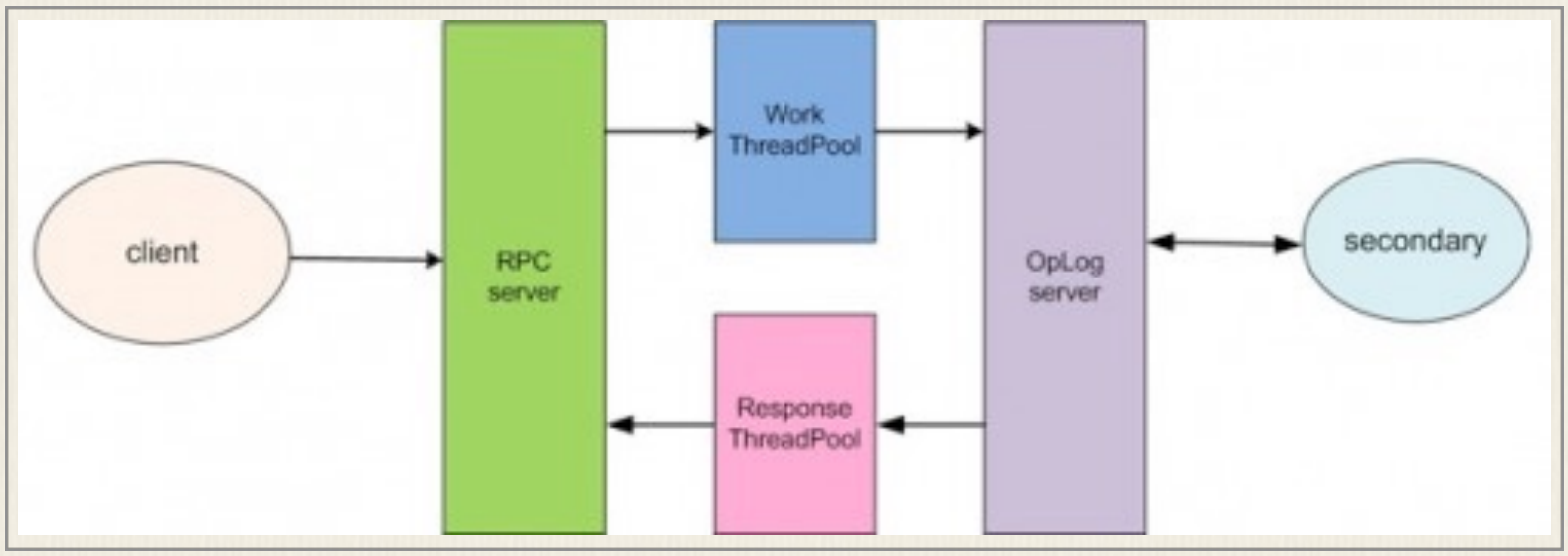


图4 写操作Pipeline简要流程图

类似中断处理程序，整个RPC的处理流程分成了上半部和下半部，上半部由Worker线程处理Request，将Operation Log提交到Oplog Server，不再阻塞等待同步Oplog成功，而是接着处理下一个Request。下半部的工作包括填充Response，将Response写入到 RPC Server ，下半部由另外一个线程池承担，通过Oplog 同步成功的消息触发。这样Worker线程源源不断地处理新的Request，写操作IOPS显著提升。由于只有同步Oplog完全成功才会返回 Response，所以数据一致性与此前的实现相同。

Group Commit

经过Pipeline优化后，写操作性能 显著提升，但我们对结果依然不太满意，想进一步提升IOPS，为上层客户提供一个更好的结果。继续Profile，发现新实现下同步Oplog吞吐量较 低，主要原因是每个写请求都导致一次主备间同步Oplog，而且这条Oplog在Primary和Secondaries上都需要同步写到磁盘。压力较高 时，将有大量的同步RPC和小片数据同步写磁盘，导致吞吐量低。通常分布式系统会使用Group Commit技术来优化这个问题，将时间上接近的Oplog组成一个Group，整个Group一次提交，吞吐量可以明

显提升。但传统的Group Commit会带来Latency的明显增加，需要在吞吐量和Latency之间做权衡。鱼与熊掌如何兼得？我们对Group Commit过程进行了适当优化，较好地解决了这个问题。

将组Group和同步Group分离，前者由一个Serialize线程承担，后 者由Sync线程完成。Serialize线程作为生产者，Sync线程作为消费者，两者通过一个队列来共享数据，当Serialize线程发现队列中大 于M个Group等待被同步的时候，将暂停组新的Group；而当Sync线程发现队列中等待的Group小于N个的时候，将唤醒Serialize线程 组新的Group，由于Serialize线程经过了一段时间的等待，积累了一批数据，此时可以组成更大的Group，同时又不会造成Latency的提 升。当整个系统负载低的时候，队列为空，Serialize无需等待，Latency很低；当系统负载较高的时候，队列中有堆积，此时暂停 Serialize，也不会增加额外的Latency，而且随后可以组成较大的Group，获得高吞吐量收益。通过对Serialize和Sync操作的 压力测试，可以确定最优化的M、N值。

细节篇

专注细节，做深做透对一个大型复杂系统而言尤为重要。在整个优化过程中，我们遇到很多富有趣味性的细节问题，经过深入挖掘后，取得了良好成果。其中具有代表性的一个例子就是Sniff的深入优化。

在 5K项目前，盘古Master冷启动时间以小时计，5K后由于规模扩张，Chunk数剧增，冷启动时间会更长。冷启动时，需要做Sniff操作，获取所有ChunkServer上Chunk的信息（Chunk的数量在十亿数量级），将这些信息汇聚到几个Map结构中，在多线程环境下Map结构的插入和更新 必须锁保护。通过锁工具Profile也证实瓶颈就在锁保护的Map上。所以缩短启动时间就转化为如何对锁保护的Map进行读写性能优化这样十分具体的细 节问题。为减少锁竞争，调整结构如图5所示。

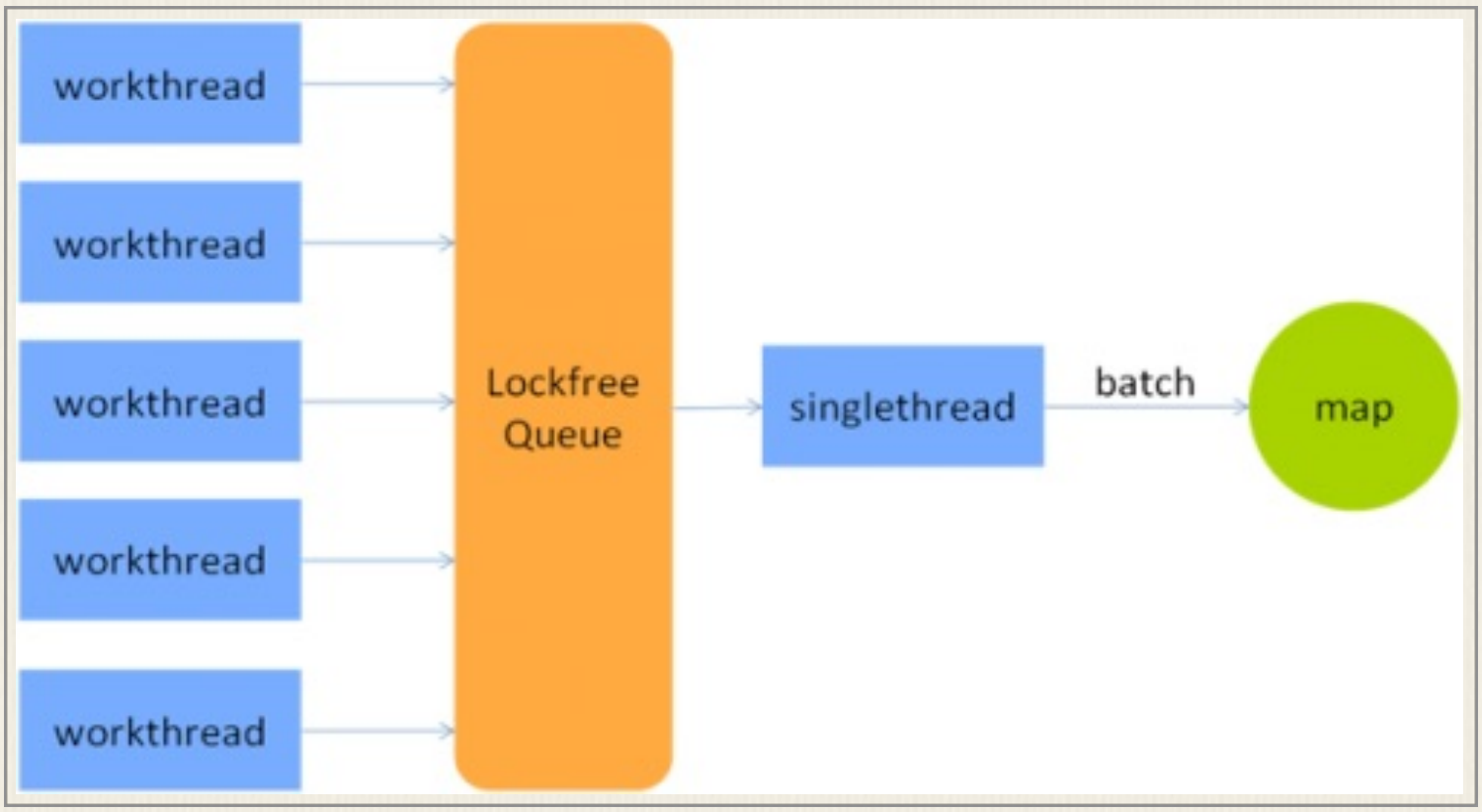


图5 Sniff优化示意图

引入一个无锁队列，所有的工作线程将数据更新到无锁队列，随后由单一线程从无锁队列中批量更新到Map，调整后Sniff期间Context Switch从40万降低到4万左右，耗时缩减到半小时，效果显著。但我们依然觉得过长，继续深挖，发现此时CAS操作异常频繁，而且单一的更新线程已经 占满一个核，继续优化感觉无从下手了，此时回过头来研究业务特点，看看能否根据业务特点来减少某些约束。功夫不负有心人，最终我们发现了一些非常有趣的细 节，即在Sniff阶段，我们基本不读取这些Map，只在Sniff完成后，第一个写请求的处理过程中需要读取Map，这意味着只要Sniff完成后Map结果保证正确即可，而在Sniff过程中，Map更新不及时导致的不准确是可接受的。根据这个细节，我们让每个工作线程在TSD（线程私有数据）中缓存一个同类型的Map，形成图6中的结构。

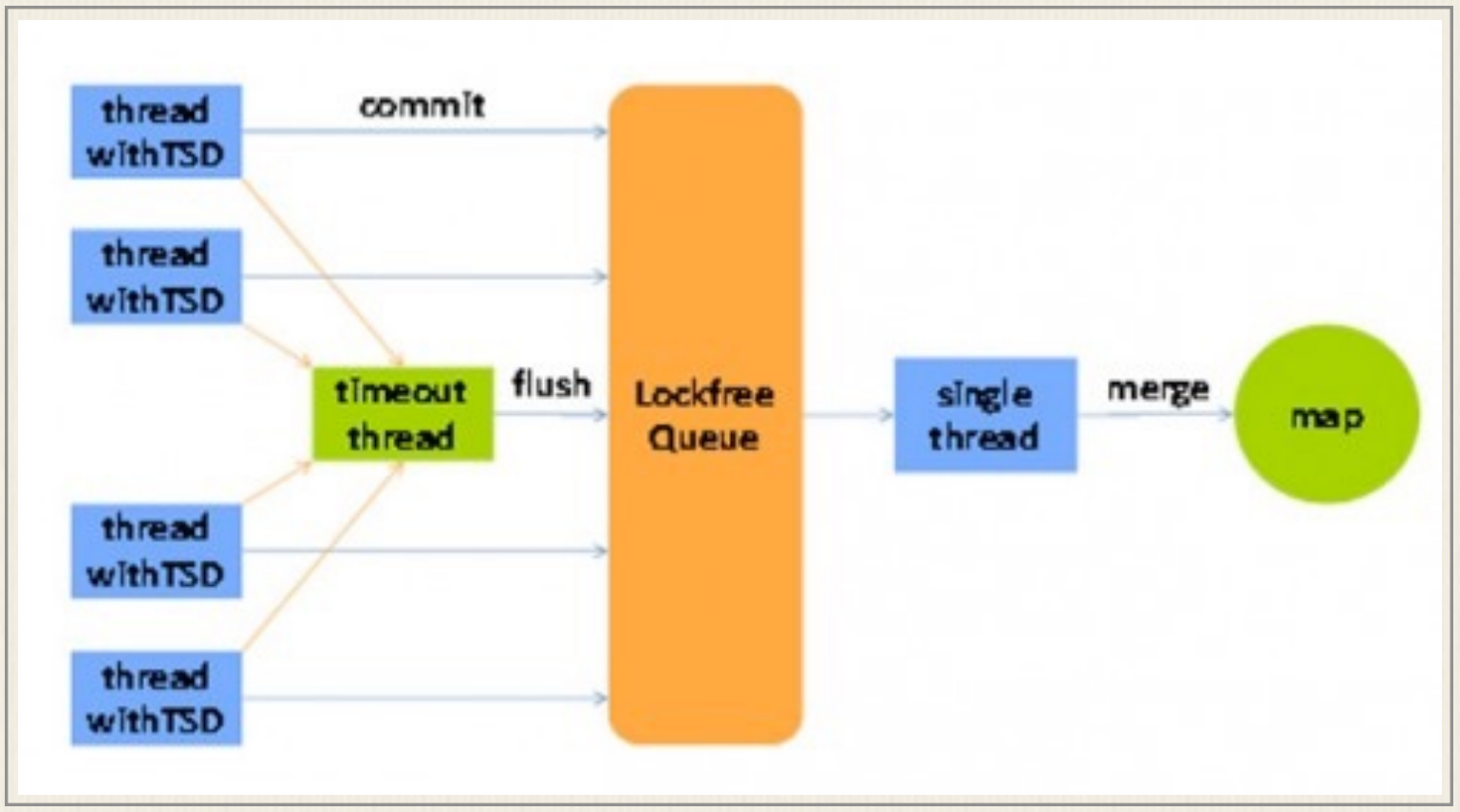


图6 Sniff深入优化示意图

每个工作线程直接将Sniff数据更新到线程私有的Map中，这个过程不需要锁保护，当TSD中汇聚的数据超过一定规模，或者数据沉淀了一定时间后再提交到无锁队列中，这样此前每个数据进一次无锁队列变成批量数据进一次队列，效率明显提升。当然这里也产生一个新问题，Map进入无锁队列是由前端 Sniff数据驱动的，当Sniff完成时，前端再无数据驱动，TSD中可能滞留了一批“沉没”数据无法提交到队列中，影响最终结果的准确性，这里我们引 入一个超时线程来解决这个问题。最终优化完成后，Sniff在数分钟内完成。

总结与展望

经过这一轮优化，Master IOPS有数倍提升，冷启动时间大幅缩减，取得了良好效果。优化过程中我们形成了如下一些共识，希望能对后续工作有所指导。

- 坚持用数据来确认瓶颈。一个点是不是瓶颈，是否需要优化，这是一个根本的方向性问题。可以大胆猜想，但一定要用数据来确认具体的瓶颈，否则方向性问题搞错了，会导致后续在非关键路径上浪费大量资源。
- 密切结合业务逻辑。很多时候系统的优化不是一个简单的计算机科学领域问题，而是与业务逻辑高度相关，结合业务逻辑特点进行优化，往往会事半功倍。
- 追求极致。在达到预期目标后，再问问自己是否达到了理论极限？是否还有潜力可挖？有时候多走一步，性能可能会有质的飞跃。

性能优化不仅仅是一项工作，更是一种锲而不舍、精益求精的态度，优化没有终点，我们一直在路上！

作者简介：吴均平，阿里云飞天高级专家，毕业于国家海洋局第二海洋研究所，曾就职于腾讯无线研发部，长期从事分布式系统研发，对分布式系统有浓厚的兴趣和一定的积累。

原文链接：http://www.csdn.net/article/2014-10-21/2822201

高性能网络编程技术

作者：jmz（360电商技术组）

如何使网络服务器能够处理数以万计的客户端连接，这个问题被称为C10K Problem。在很多系统中，网络框架的性能直接决定了系统的整体性能，因此研究解决高性能网络编程框架问题具有十分重要的意义。

1. 网络编程模型

在C10K Problem中，给出了一些常见的解决大量并发连接的方案和模型，在此根据自己理解去除了一些不实际的方案，并做了一些整理。

1.1、PPC/TPC模型

典型的Apache模型（Process Per Connection，简称PPC），TPC（Thread Per Connection）模型，这两种模型思想类似，就是让每一个到来的连接都一边自己做事直到完成。只是PPC是为每个连接开了一个进程，而TPC开了一个线程。可是当连接多了之后，如此多的进程/线程切换需要大量的开销；这类模型能接受的最大连接数都不会高，一般在几百个左右。

1.2、异步网络编程模型

异步网络编程模型都依赖于I/O多路复用模式。一般地,I/O多路复用机制都依赖于一个事件多路分离器(Event Demultiplexer)。**分离器对象可将来自事件源的I/O事件分离出来，并分发到对应的read/write事件处理器(Event Handler)**。开发人员预先注册需要处理的事件及其事件处理器（或回调函数）；事件分离器负责将请求事件传递给事件处理器。两个与事件分离器有关的模式是Reactor和Proactor。Reactor模式采用同步IO，而Proactor采用异步IO。

在Reactor中，事件分离器负责等待文件描述符或socket为读写操作准备就绪，然后将就绪事件传递给对应的处理器，最后由处理器负责完成实际的读写工作。

而在Proactor模式中，处理器--或者兼任处理器的事件分离器，只负责发起异步读写操作。**IO操作本身由操作系统来完成**。传递给操作系统的参数需要包括用户定义的数据缓冲区地址和数据大小，操作系统才能从中得到写出操作所需数据，或写入从socket读到的数据。事件分离器捕获IO操作完成事件，然后将事件传递给对应处理器。

l 在Reactor中实现读：

- 注册读就绪事件和相应的事件处理器

- 事件分离器等待事件

- 事件到来，激活分离器，分离器调用事件对应的处理器

- 事件处理器完成实际的读操作，处理读到的数据，注册新事件，然后返还控制权

l 在Proactor中实现读：

- 处理器发起异步读操作（注意：操作系统必须支持异步IO）。在这种情况下，处理器无视IO就绪事件，它关注的是完成事件。

- 事件分离器等待操作完成事件

- 在分离器等待过程中，操作系统利用并行的内核线程执行实际的读操作，并将结果数据存入用户自定义缓冲区，最后通知事件分离器读操作完成。

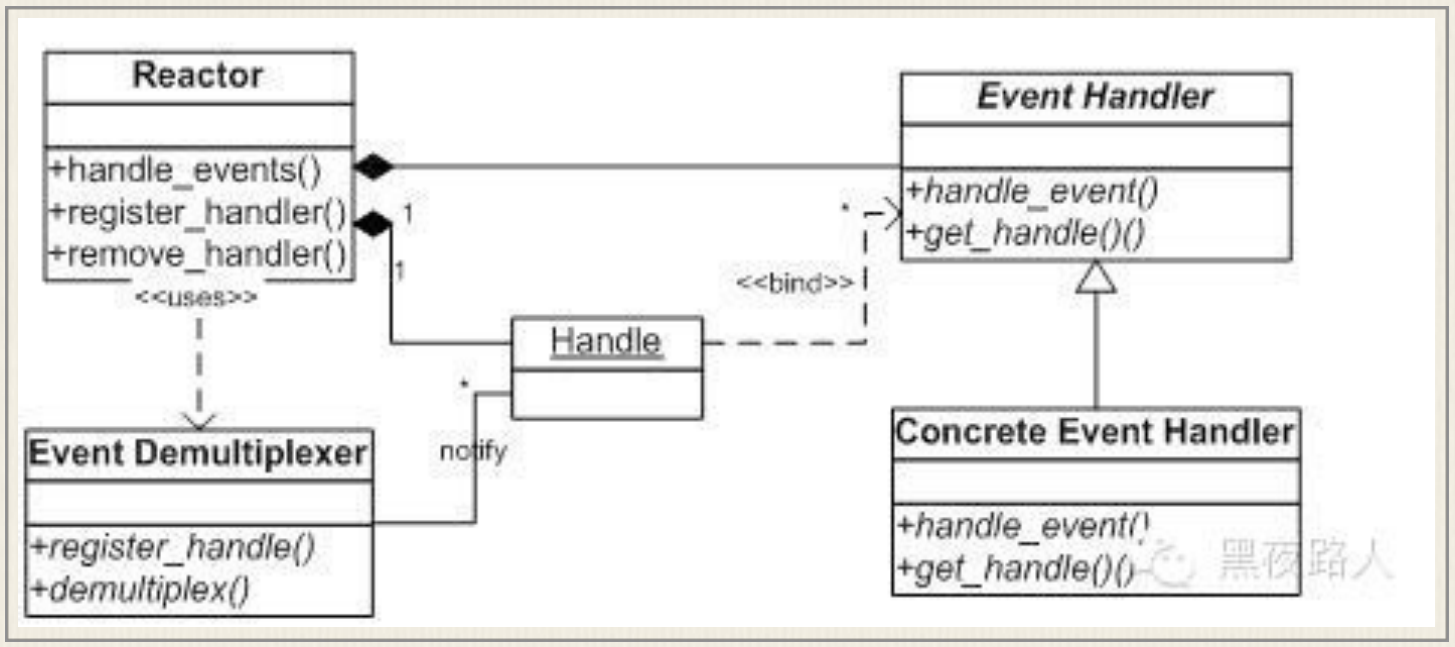
- 事件分离器呼唤处理器。

- 事件处理器处理用户自定义缓冲区中的数据，然后启动一个新的异步操作，并将控制权返回事件分离器。

可以看出，两个模式的相同点，都是对某个IO事件的事件通知(即告诉某个模块，这个IO操作可以进行或已经完成)。在结构上，两者也有相同 点：demultiplexor负责提交IO操作(异步)、查询设备是否可操作(同步)，然后当条件满足时，就回调handler；不同点在于，**异步情况下(Proactor)，当回调handler时，表示IO操作已经完成；同步情况下(Reactor)，回调handler时，表示IO设备可以进行某个操作(can read or can write)**。

1.2.1 Reactor模式框架

使用Proactor模式需要操作系统支持异步接口，因此在日常中比较常见的是Reactor模式的系统调用接口。使用Reactor模型，必备的几个组件：事件源、Reactor框架、多路复用机制和事件处理程序，先来看看Reactor模型的整体框架，接下来再对每个组件做逐一说明。



I 事件源

Linux上是文件描述符，Windows上就是Socket或者Handle了，这里统一称为“句柄集”；程序在指定的句柄上注册关心的事件，比如I/O事件。

I event demultiplexer——事件多路分发机制

Ø 由操作系统提供的I/O多路复用机制，比如select和epoll。

Ø 程序首先将其关心的句柄（事件源）及其事件注册到event demultiplexer上；

Ø 当有事件到达时，event demultiplexer会发出通知“在已经注册的句柄集中，一个或多个句柄的事件已经就绪”；

Ø 程序收到通知后，就可以在非阻塞的情况下对事件进行处理了。

l Reactor——反应器

Reactor，是事件管理的接口，内部使用event demultiplexer注册、注销事件；并运行事件循环，当有事件进入“就绪”状态时，调用注册事件的回调函数处理事件。

一个典型的Reactor声明方式

```
class Reactor {
    public:
        int register_handler(Event_Handler *pHandler, int event);
        int remove_handler(Event_Handler *pHandler, int event);
        void handle_events(timeval *ptv);
        // ...
    };
```

l Event Handler——事件处理程序

事件处理程序提供了一组接口，每个接口对应了一种类型的事件，供Reactor在相应的事件发生时调用，执行相应的事件处理。通常它会绑定一个有效的句柄。

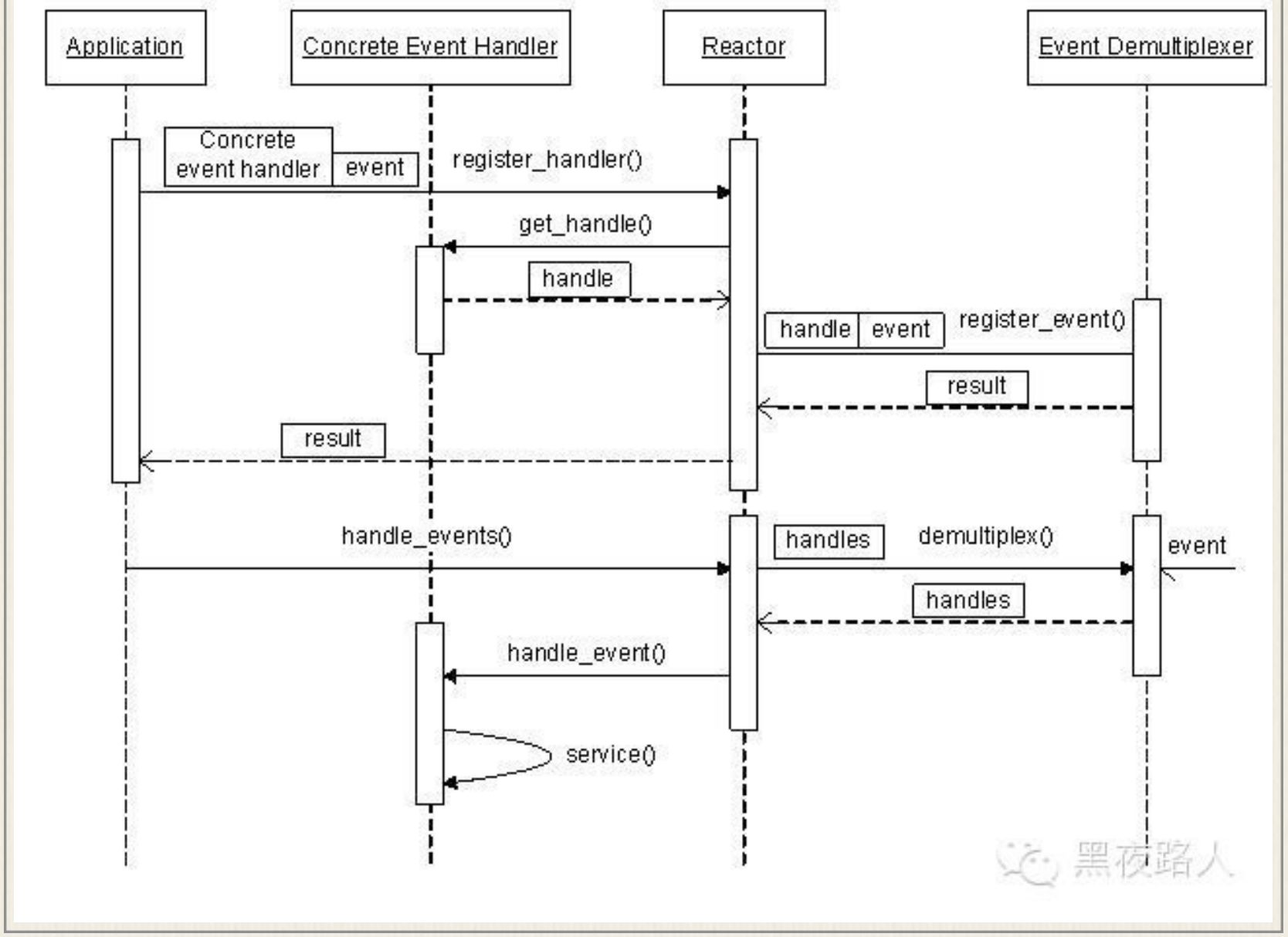
下面是两种典型的Event Handler类声明方式，二者互有优缺点。

```
class Event_Handler {
public:
```

```
    virtual void handle_read() = 0;
    virtual void handle_write() = 0;
    virtual void handle_timeout() = 0;
    virtual void handle_close() = 0;
    virtual HANDLE get_handle() = 0;
    // ...
};
class Event_Handler {
public:
    // events maybe read/write/timeout/close .etc
    virtual void handle_events(int events) = 0;
    virtual HANDLE get_handle() = 0;
    // ...
};
```

1.2.2 Reactor事件处理流程

前面说过Reactor将事件流“逆置”了，使用Reactor模式后，事件控制流可以参见下面的序列图。



1.3 Select，poll和epoll

在Linux环境中，比较常见的I/O多路复用机制就是Select，poll和epoll，下面对这三种机制进行分析和比较，并对epoll的使用进行介绍。

1.3.1 select模型

1. 最大并发数限制，因为一个进程所打开的FD（文件描述符）是有限制的，由FD_SETSIZE设置，默认值是1024/2048，因此Select模型的最大并发数就被相应限制了。

2. 效率问题，select每次调用都会线性扫描全部的FD集合，这样效率就会呈现线性下降，把FD_SETSIZE改大的后果就是所有FD处理都慢慢来

3. 内核/用户空间 内存拷贝问题，如何让内核把FD消息通知给用户空间呢？在这个问题上select采取了内存拷贝方法。

```
int res = select(maxfd+1, &readfds, NULL, NULL, 120);
if (res > 0) {
    for (int i = 0; i < MAX_CONNECTION; i++) {
        if (FD_ISSET(allConnection[i],&readfds)) {
            handleEvent(allConnection[i]);
        }
    }
}
```

1.3.2 poll模型

基本上效率和select是相同的，select缺点的2和3都没有改掉。

1.3.3 epoll模型

1. Epoll没有最大并发连接的限制，上限是最大可以打开文件的数目，这个数字一般远大于2048, 一般来说这个数目和系统内存关系很大，具体数目可以cat /proc/sys/fs/file-max察看。

2. 效率提升，Epoll最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，应用程序就能直接定位到事件，而不必遍历整个FD集合，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll。

```
int res = epoll_wait(epfd, events, 20, 120);
for(int i = 0; i < res; i++) {
    handleEvent(events[n]);
}
```

3. 内存拷贝，Epoll在这点上使用了“共享内存”，这个内存拷贝也省略了。

1.3.4 使用epoll

Epoll的接口很简单，只有三个函数，十分易用。

int epoll_create(int size);

生成一个epoll专用的文件描述符，其实是申请一个内核空间，用来存放你想关注的socket fd上是否发生以及发生了什么事件。size就是你在这个Epoll fd上能关注的最大socket fd数，大小自定，只要内存足够。

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);

控制某个Epoll文件描述符上的事件：注册、修改、删除。其中参数epfd是epoll_create()创建Epoll专用的文件描述符。相对于select模型中的FD_SET和FD_CLR宏。

int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);

等待I/O事件的发生；参数说明：

Ø epfd:由epoll_create()生成的Epoll专用的文件描述符；

Ø epoll_event:用于回传代处理事件的数组；

Ø maxevents:每次能处理的事件数；

Ø timeout:等待I/O事件发生的超时值；

Ø 返回发生事件数。

上面讲到了Reactor的基本概念、框架和处理流程，并对基于Reactor模型的select，poll和epoll进行了比较分析后，再来对比看网络编程框架就会更容易理解了。

2. Libeasy网络编程框架

Libeasy底层使用的是Libev事件库，在分析Libeasy代码前，首先对Libev有相关了解。

2.1 Libev简介

l Libev是什么?

Libev is an event loop: you register interest in certain events (such as a file descriptor being readable or a timeout occurring), and it will manage these event sources and provide your program with events.

Libev是一个event loop：向libev注册感兴趣的events，比如Socket可读事件，libev会对所注册的事件的源进行管理，并在事件发生时触发相应的程序。通过event watcher来注册事件

l libev定义的watcher类型：

Ø ev_io // io 读写类型watcher

Ø ev_timer // 定时器 类watcher

Ø ev_periodic

Ø ev_signal

Ø ev_child

Ø ev_stat

Ø ev_idle

Ø ev_prepare

Ø ev_check

Ø ev_embed

Ø ev_fork

Ø ev_cleanup

Ø ev_async // 线程同步信号watcher

在libev中watcher还能支持优先级

2.1.1 libev使用

下面以一个简单例子程序说明libev的使用。这段程序实现从标准输入异步读取数据，5.5秒内没有数据到来则超时的功能。

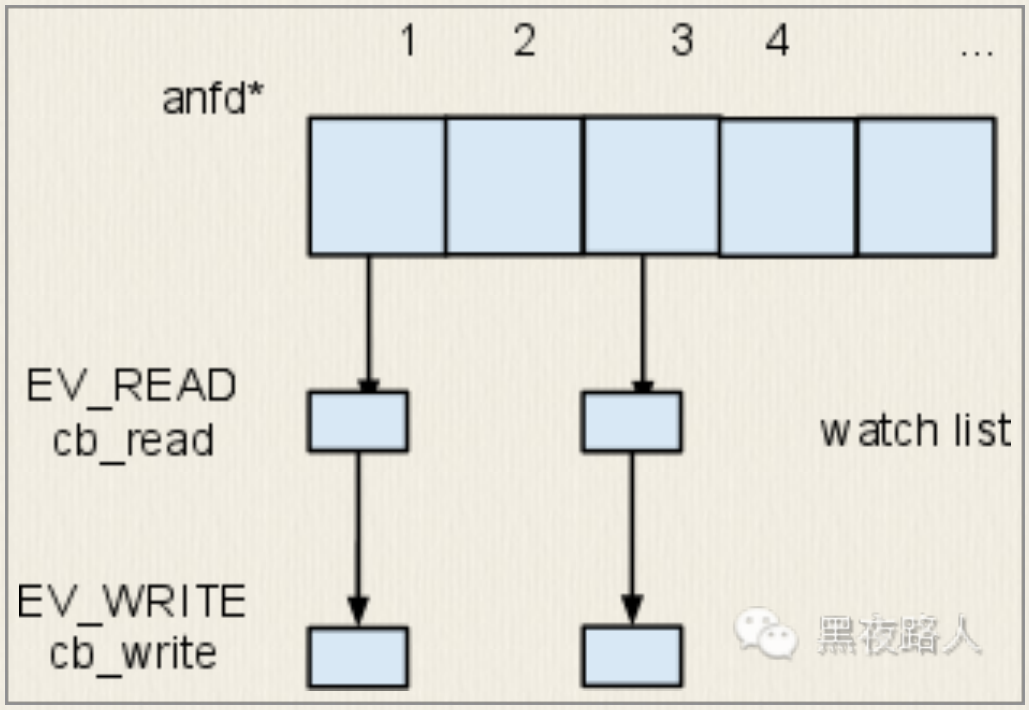
```
#include <ev.h>
#include <stdio.h>
ev_io stdin_watcher;
ev_timer timeout_watcher;
// all watcher callbacks have a similar signature
// this callback is called when data is readable on stdin
static void stdin_cb (EV_P_ ev_io *w, int revents) {
  puts ("stdin ready");
  // for one-shot events, one must manually stop the watcher
  // with its corresponding stop function.
  ev_io_stop (EV_A_ w);
  // this causes all nested ev_run's to stop iterating
  ev_break (EV_A_ EVBREAK_ALL);
}
// another callback, this time for a time-out
static void timeout_cb (EV_P_ ev_timer *w, int revents) {
  puts ("timeout");
  // this causes the innermost ev_run to stop iterating
```

```
    ev_break (EV_A_ EVBREAK_ONE);
  }
  int main (void) {
    // use the default event loop unless you have special needs
    struct ev_loop *loop = EV_DEFAULT;
    // initialise an io watcher, then start it
    // this one will watch for stdin to become readable
    ev_io_init (&stdin_watcher, stdin_cb, /*STDIN_FILENO*/ 0, EV_REA
D);
    ev_io_start (loop, &stdin_watcher);
    // initialise a timer watcher, then start it
    // simple non-repeating 5.5 second timeout
    ev_timer_init (&timeout_watcher, timeout_cb, 5.5, 0.);
    ev_timer_start (loop, &timeout_watcher);
    // now wait for events to arrive
    ev_run (loop, 0);
    // break was called, so exit
    return 0;
  }
```

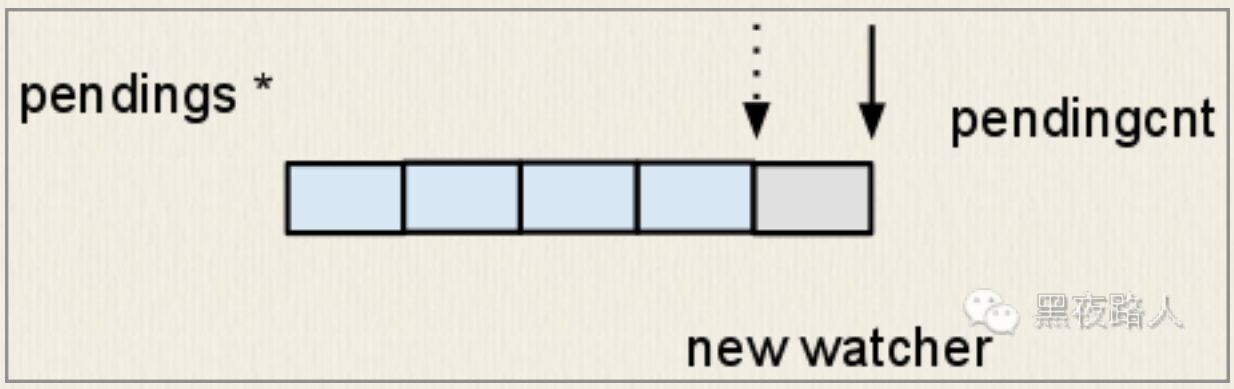
2.1.2 Libev和Libevent比较

libevent和libev架构近似相同，对于非定时器类型，libevent使用双向链表管理，而libev则是使用数组来 管理。如我们所知，新的fd总是系统可用的最

小fd，所以这个长度可以进行大小限制的，我们用一个连续的数组来存储fd/watch 信息。如下图，我们 用anfd[fd]就可以找到对应的fd/watcher 信息，当然可能遇到anfd超出我们的buffer长度情形，这是我们用类似relloc 的 函数来做数组迁移、扩大容量，但这种概率是很小的，所以其对系统性能的影响可以忽略不计。



我们用anfd[fd]找到的结构体中，有一个指向io_watch_list的头指针，以epoll为例，当epoll_wait返回一个 fd_event时 ，我们就可以直接定位到对应fd的watch_list，这个watch_list的长度一般不会超过3 ，fd_event会有一 个导致触发的事件，我们用这个事件依次和各个watch注册的event做 “&” 操作, 如果不为0，则把对应的watch加入到待处理队列pendings中(当我们启用watcher优先级模式时，pendings是个2维数组，此时仅考虑普通模式)所以我们可以看到，这个操作是非常非常快。



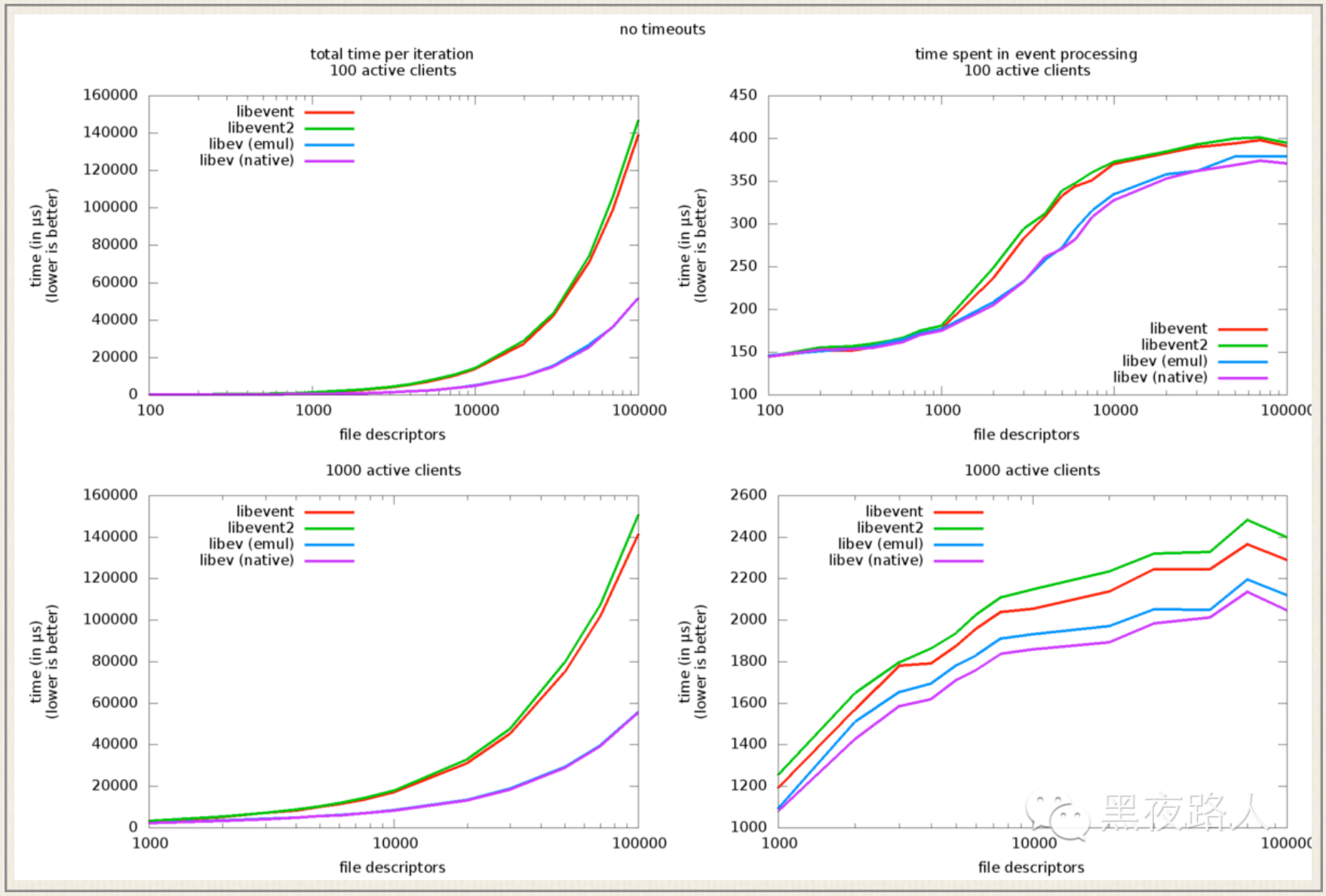
再看添加watch的场景，把watch插入到相应的链表中，这个操作也是直接定位，然后在fdchange队列中，加入对应的fd（如果这个fd已经被添加过，则不会发生这一步，我们通过anfd[fd]中一个bool 值来判断）

注 意，假如我们在某个fd上已经有个watch 注册了read事件，这时我们又再添加一个watch，还是read 事件，但是不同的回调函数，在此种情 况下，我们不应该调用epoll_ctrl 之类的系统调用，因为我们的events集合是没有改变的，所以为了达到这个目的，anfd[fd]结构体 中，还有一个events事件，它是原先的所有watcher的事件的“|”操作，向系统的epoll从新添加描述符的操作是在下次事件迭代开始前进行的，当我们依次扫描fdchangs，找到对应的anfd结构，如果发现先前的events与当前所有的watcher的“|”操作结果不等，则表示我们需要调用epoll_ctrl之类的函数来进行更改，反之不做操作，作为一条原则，在调用系统调用前，我们已经做了充分的检查，确保不进行多余的系统调用。

再来看删除和更新一个watcher造作，基于以上分析，这个操作也是近乎O(1) 的，当然，如果events事件更改，可能会发生一次系统调用。

所以我们对io watcher的操作，在我们的用户层面上，几乎总是是O(1)的复杂度，当然如果牵涉到epoll 文件结构的更新，我们的系统调用 epoll_ctrl 在内核中还是 O(lgn)的复杂度，但我们已经在我们所能掌控的范围内做到最好了。

2.1.3 性能测试对比



结论：

The cost for setting up or changing event watchers is clearly much higher for libevent than for libev，详细性能对比测试参考这http://libev.schmorp.de/bench.html

2.2 libeasy

2.2.2 Server端使用

1、启动流程

eio_ = easy_eio_create(eio_, io_thread_count_);

easy_eio_create(eio_, io_thread_count_)做了如下几件事：

1. 分配一个easy_pool_t的内存区，存放easy_io_t对象

2. 设置一些tcp参数，比如tcp_nodelay（tcp_cork），cpu亲核性等参数

3. 分配线程池的内存区并初始化

4. 对每个线程构建client_list，client_array, 初始化双向链表 conn_list session_list request_list

5. 设置listen watcher的ev回调函数为easy_connection_on_listen

6. 调用easy_baseth_init初始化io线程

easy_listen_t* listen = easy_connection_add_listen(eio_, NULL, port_, &handler_);

1. 从eio->pool中为easy_listen_t和listen watcher（在这里listen的watcher数默认为2个）分配空间

2. 开始监听某个地址

3. 初始化每个read_watcher

4. 关注listen fd的读事件，设置其回调函数 easy_connection_on_accep（在这里仅仅是初始化read_watcher, 还没有激活，激活在每个IO线程启动调用easy_io_on_thread_start的时候做。一旦激活后，当有连接到来的时候，触发easy_connection_on_accept）

rc = easy_eio_start(eio_);

1. 调用pthread_create启动每个io线程，线程执行函数 easy_io_on_thread_start，在easy_io_on_thread_start中

a) 设置io线程的cpu亲核性sched_setaffinity

b) 如果不是listen_all或者只有一个线程，则发出ev_async_send唤醒下一个线程的listen_watcher(实现连接请求的负载均衡)

2. 线程执行ev_run

easy_eio_wait(eio_);

调用pthead_join等待线程结束

2、处理流程

l 当连接到来时触发easy_connection_on_accept

1. 调用accept获得连接fd，构建connection（easy_connection_new），设置非阻塞，初始化connection参数和read、write、timeout的watcher

2. 切换listen线程，从自己切换到下一个io线程，调用ev_async_send激活下一个io线程的listen_watcher，实现负载均衡

3. 将connection加入到线程的connected_list线程列表中，并开启该连接上的read、write、timeout的watcher

l 当数据包到来时触发easy_connection_on_readable回调函数

1. 检 查当前IO线程同时正在处理的请求是否超过EASY_IOTH_DOING_REQ_CNT(8192)，当前连接上的请求数是否超过EASY_CONN_DOING_REQ_CNT（1024），如果超过，则调用easy_connection_destroy(c)将连接销毁 掉, 提供了一种负载保护机制

2. 构建message空间

3. 调用read读取socket数据

4. 作为服务端调用easy_connection_do_request

a) 从message中解包

b) 调用easy_connection_recycle_message看是否需要释放老的message，构建新的message空间

c) 调用hanler的process处理数据包，如果返回easy_ok则调用easy_connection_request_done

d) 对发送数据进行打包

e) 对返回码是EASY_AGAIN的request将其放入session_list中

f) 对返回码是EASY_OK的request将其放入request_done_list中，更新统计计数

g) 统计计数更新

h) 调用easy_connection_write_socket发送数据包

i) 调用easy_connection_evio_start中ev_io_start(c->loop, &c->read_watcher);开启该连接的读watcher

j) 调用easy_connection_redispatch_thread进行负载均衡

如 果负载均衡被禁或者该连接的message_list和output不为空，则直接返回，否则调用easy_thread_pool_rr从线程池中选择 一个io线程，将该连接从原来io线程上移除（停止读写timeout 的watcher），将该连接加入到新的io线程中的conn_list中，调用 ev_async_send唤醒新的io线程，在easy_connection_on_wakeup中调用 easy_connection_evio_start将该连接的read、write、timeou的watcher再打开。

l 当socket可写时触发easy_connection_on_writable回调函数：

1. 调用easy_connection_write_socket写数据

2. 如果没有数据可写，将该连接的write_watcher停掉

2.2.3 客户端使用

libeasy作为客户端时，将每个发往libeasy服务器端的请求包封装成一个session(easy_session_t)，客户端将这个session放入连接的队列中然后返回，随后收到包后，将相应的session从连接的发送队列中删除。详细流程如下：

easy_session_t *easy_session_create(int64_t asize)

这个函数主要就做了一件事分配一个内存池easy_pool_t，在内存池头部放置一个easy_session_t，剩下部分存放实际的数据包Packet，然后将session的type设置为EASY_TYPE_SESSION。

异步请求

int easy_client_dispatch(easy_io_t *eio, easy_addr_t addr, easy_session_t *s)

1. 根据socket addr从线程池中选择一个线程,将session加入该线程的session_list，然后将该线程唤醒

2. 线程唤醒后调用easy_connection_send_session_list

a) 其 中首先调用easy_connection_do_client，这里首先在该线程的client_list中查找该addr的client，如果没找 到，则新建一个client，初始化将其加入client_list，如果该client的connect未建立，调用 easy_connection_do_connect建立该连接，然后返回该连接

b) easy_connection_do_connect 中首先创建一个新的connection结构，和一个socket，设置非阻塞，并调用connect进行连接，初始化该连接的read、write、 timeout watcher（连接建立前是write，建立后是read）

c) 调用easy_connection_session_build，其中调用encode函数对 数据包进行打包，调用 easy_hash_dlist_add(c->send_queue, s->packet_id, &s->send_queue_hash, &s->send_queue_list) 将这个session添加到连接的发送队列中。这个函数将session添加到发送队列的同时，同时将相应的项添加到hash表的相应的bucket的链 表头

d) 开启timeout watcher

e) 调用easy_connection_write_socket发送数据包

l 当回复数据包到达触发easy_connection_on_readable回调函数时

1. 初始化一个easy_message_t存放数据包

2. 从内核缓冲区读入数据到应用层输入缓冲区中，然后调用easy_connection_do_response进行处理

a) 先解包，将该packet_id数据包从发包队列中删除，更新统计信息，停止timeout watcher，

b) 如果是同步请求，则调用session的process函数，从而调用easy_client_wait_process函数，唤醒客户端接收数据包

I 当超时时间到还没有收到回复数据包时触发easy_connection_on_timeout_mesg回调函数

1. 从发送队列中删除请求数据包

2. 调用session的process函数，从而调用easy_client_wait_process函数，唤醒客户端接

3. 释放此连接

同步请求

void *easy_client_send(easy_io_t *eio, easy_addr_t addr, easy_session_t *s)

同步请求是通过异步请求实现的，easy_client_send方法封装了异步请求接口easy_client_dispatch

1. easy_client_send将session的process置为easy_client_wait_process方法

2. 初始化一个easy_client_wait_t wobj

3. 调用easy_client_dispatch方法发送异步请求

4. 客户端调用wait在wobj包装的信号量上等待

5. 当这个请求收到包的时候触发session的process函数，回调easy_client_wait_process方法，其中会给wobj发送信号唤醒客户端，返回session封装的请求的ipacket

2.2.4 特性总结

1. 多个IO线程/epoll，大大提升了数据包处理性能，特别是处理小数据包的性能

针对多核处理器，libeasy使用多个IO线程来充分发挥处理器性能，提升IO处理能力。特别是针对小数据包IO处理请求数较多的情况下，性能提升十分明显。

2. 短任务和长任务区分，处理短任务更加高效（编码了内存拷贝，线程切换）

同步处理

对于短任务而言，调用用户process回调函数返回EASY_OK的数据包直接被加入该连接的发送队列，发送给客户端，这样避免了数据包的内存拷贝和线程切换开销。

异步处理

对于耗时较长的长任务而言，如果放在网络库的IO线程内执行，可能会阻塞住IO线程，所以需要异步处理。

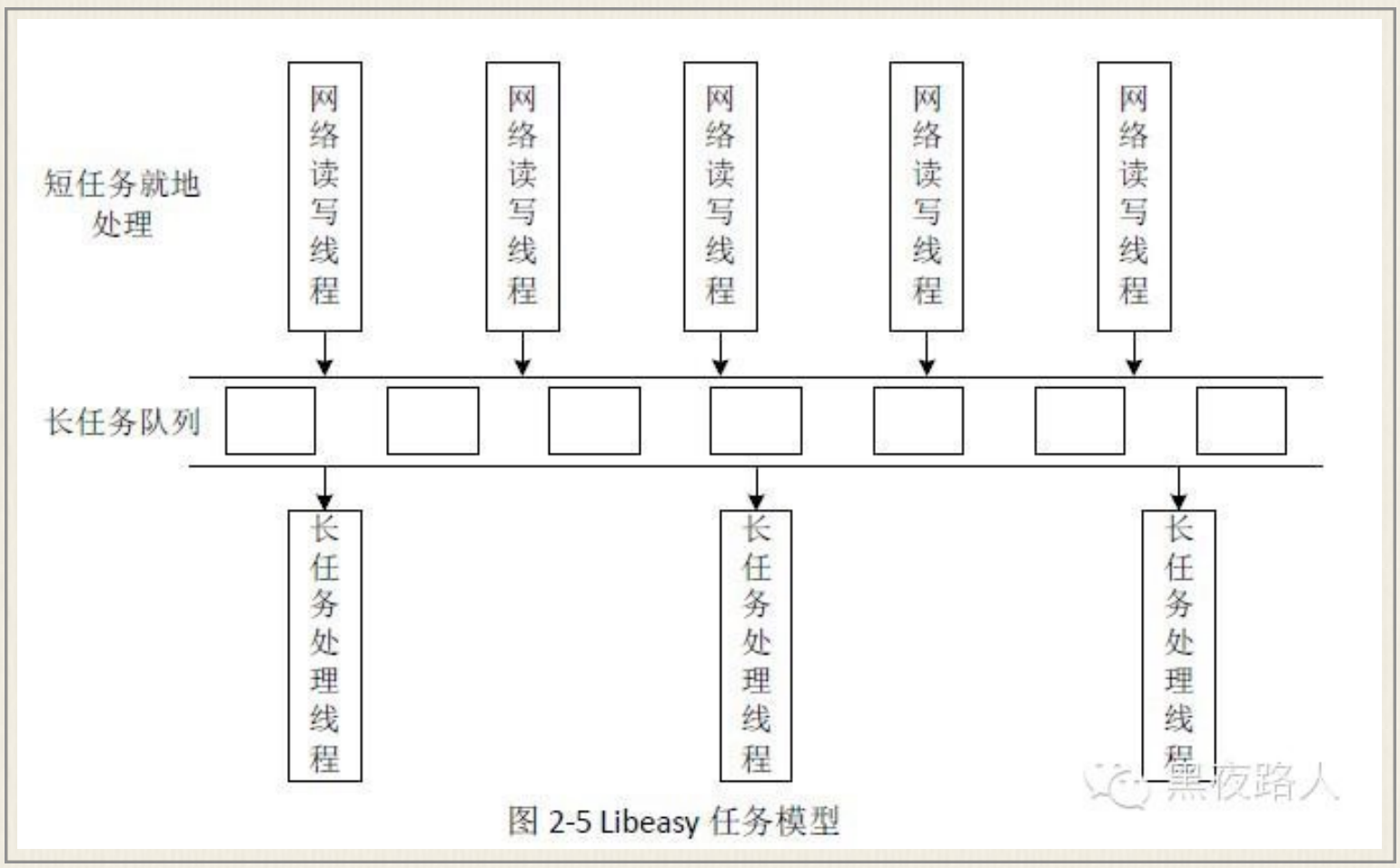


图 2-5 Libeasy 任务模型

3. 应用线程CPU亲核性，避免线程调度开销，提升处理性能

开启亲核特性将线程与指定CPU核进行绑定，避免了线程迁移导致的CPU cache失效，同时它允许我们精确控制线程和cpu核的关系，从而根据需要划分CPU核的使用。

sched_setaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask)

该 函数设置进程为pid的这个进程,让它运行在mask所设定的CPU上.如果pid的值为0,则表示指定的是当前进程,使当前进程运行在mask所设定的 那些CPU上.第二个参数cpusetsize是mask所指定的数的长度.通常设定为sizeof(cpu_set_t).如果当前pid所指定的进程 此时没有运行在mask所指定的任意一个CPU上,则该指定的进程会从其它CPU上迁移到mask的指定的一个CPU上运行.

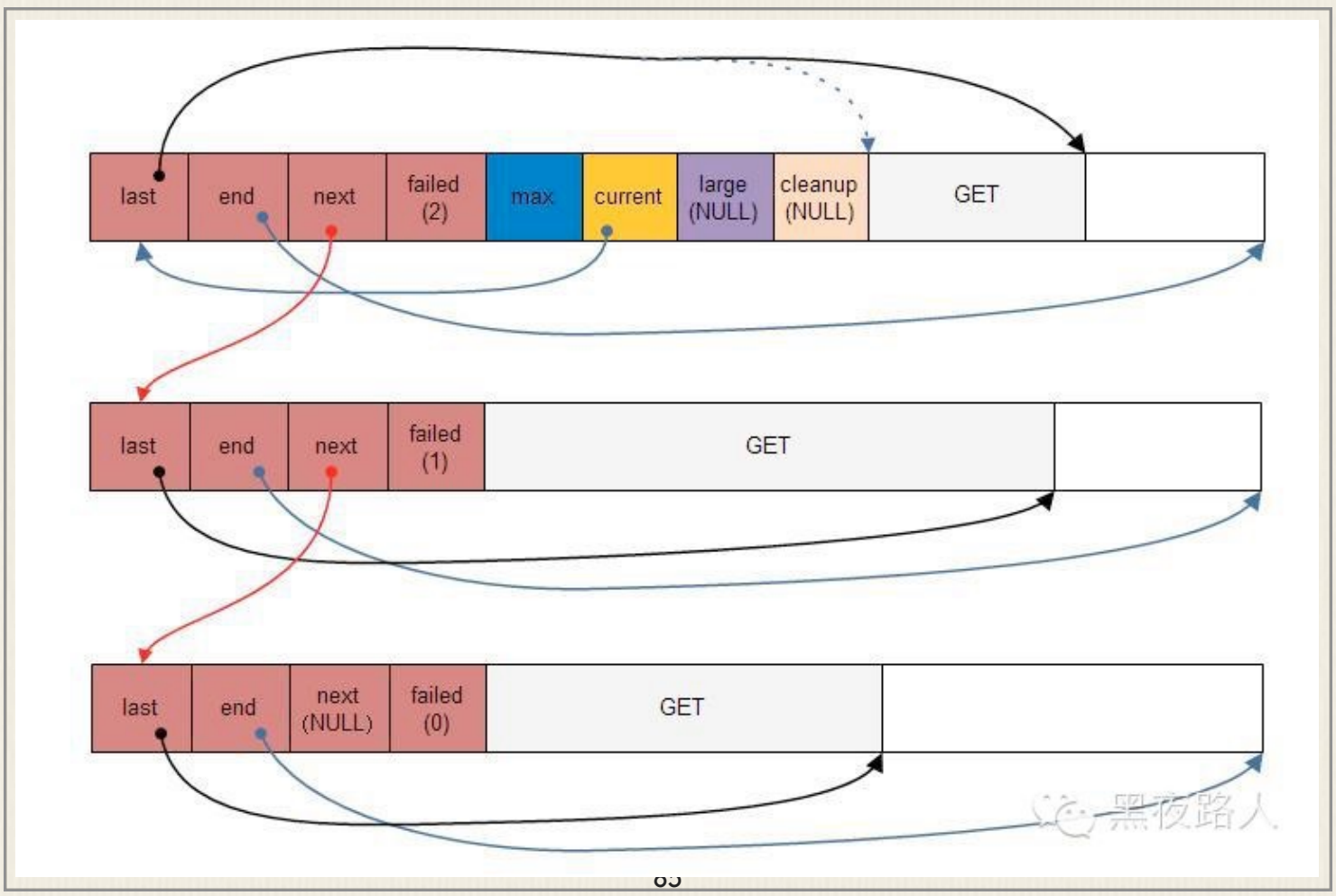
4. 内存管理，减少小内存申请开销，避免内存碎片化

Libeasy的内存管理和ng-inx一致，有兴趣的可以去学习下，下面大致介绍其思想。

1) 创建一个内存池

2) 分配小块内存(size <= max)

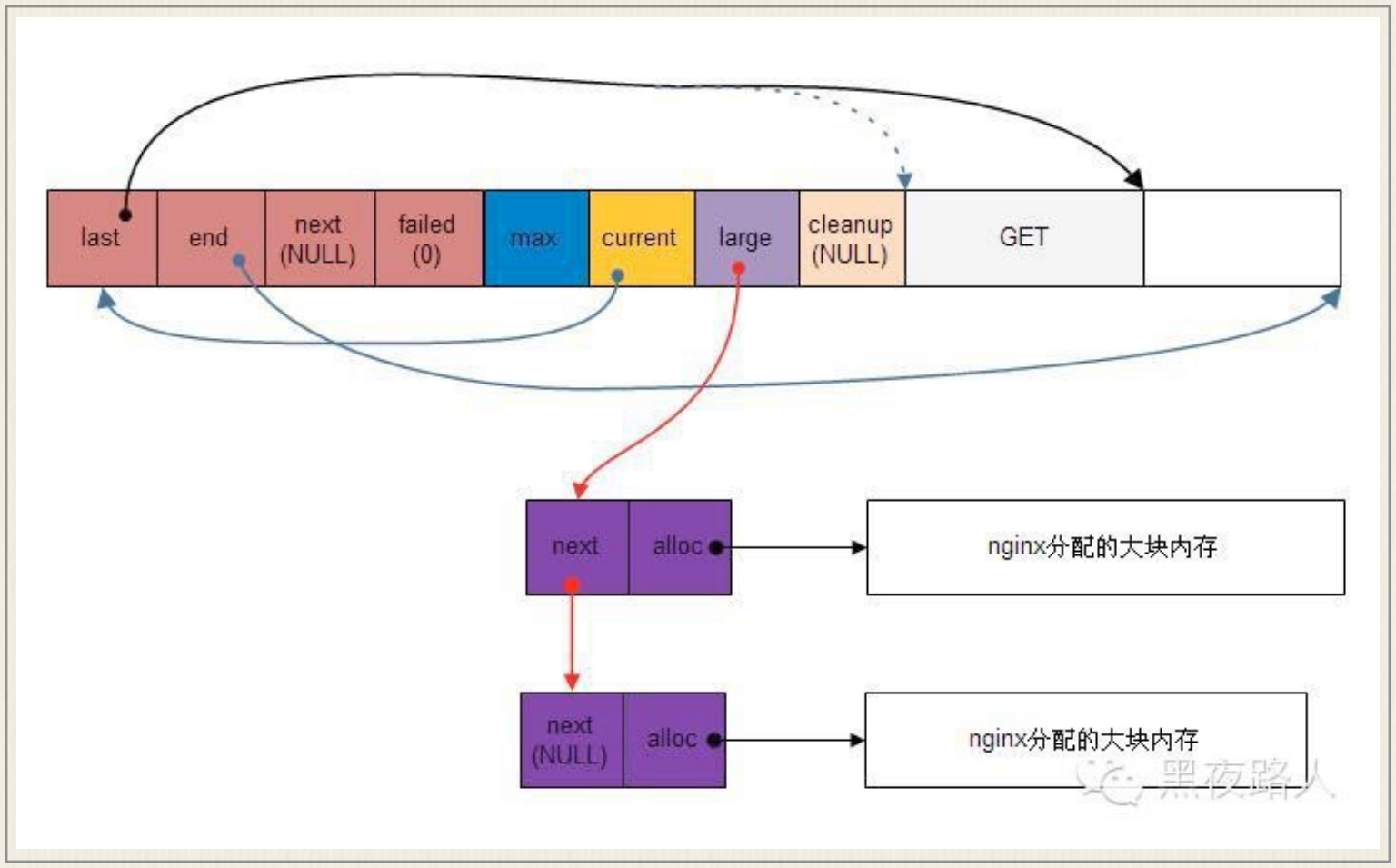
小块内存分配模型:



上图这个内存池模型是由上3个小内存池构成的，由于第一个内存池上剩余的内存不够分配了，于是就创建了第二个新的内存池，第三个内存池是由于前面两个内存池的剩余部分都不够分配，所以创建了第三个内存池来满足用户的需求。由图可见：所有的小内存池是由一个单向链表维护在一起的。这里还有两个字段需要关注，failed和current字段。failed表示的是当前这个内存池的剩余可用内存不能满足用户分配请求的次数，如果下一 个内存池也不能满足，那么它的failed也会加1，直到满足请求为止（如果没有现成的内存池来满足，会再创建一个新的内存池）。current字段会随 着failed的增加而发生改变，如果current指向的内存池的failed达到了一个阈值，current就指向下一个内存池了。

3)、大块内存的分配(size > max)

大块内存的分配请求不会直接在内存池上分配内存来满足，而是直接向操作系统申请这么一块内存（就像直接使用malloc分配内存一样），然后将这块内存挂到内存池头部的large字段下。内存池的作用在于解决小块内存池的频繁申请问题，对于这种大块内存，是可以忍受直接申请的。同样，用图形展示大块内存申请模型：



4)、内存释放

nginx 利用了web server应用的特殊场景来完成；一个web server总是不停的接受connection和request，所以nginx就将内 存池分了不同的等级，有进程级的内存池、connection级的内存池、request级的内存池。也就是说，创建好一个worker进程的时候，同时 为这个worker进程创建一个内存池，待有新的连接到来后，就在worker进程的内存池上为该连接创建起一个内存池；连接上到来一个request 后，又在连接的内存池上为request创建起一个内存池。这样，在request被处理完后，就会释放request的整个内存池，连接断开后，就会释 放连接的内存池。

5)、总结

通过内存的分配和释放可以看出，nginx只是将小块内存的申请聚集到一起申请(内存池)，然后一起释放，避免了频繁申请小内存，降低内存碎片的产生等问题。

5. 网络流量自动负载均衡，充分发挥多核性能

1、在连接到来时，正在listen的IO线程接受连接，将其加入本线程的连接队列中，之后主动唤醒下一个线程执行listen。通过切换listen线程来使每个线程上处理的连接数大致相同。

2、每一个连接上的流量是不同的，因此在每次有读写请求，计算该线程上近一段时间内请求速率，触发负载均衡，将该连接移动到其它线程上，使每个线程处理的IO请求数大致相同。

6. 将encode和decode接口暴露给应用层，实现网络编程框架与协议的分离

Libeasy将网络数据包打包解包接口暴露给应用层，由用户定义数据包内容的格式，实现了网络编程框架与协议的分离，能够支持http等其他协议类型，格式更改更加方便。

7. 底层采用libev，对于事件的注册和更改速度更快

原文链接：http://blog.csdn.net/heiyeshuwu/article/details/40508683

深入浅出Session攻击方式之一 – 固定会话ID

作者：360weboy

随着php语言的流行，出现了无数的使用php开发的web应用程序，这吸引了大批的攻击者开始寻找，攻击有安全漏洞的php应用程序。 所以，程序的安全问题得到了越来越多的关注。做为一个专业的php开发者，必须要重视安全问题。

接 下来，我们来谈谈攻击session的方法之一 - 固定会话ID。 这种攻击方式的核心要点就是让合法用户使用攻击者预先设定的session ID来访问被攻击的应用程序，一旦用户的会话ID被成功固定，攻击者就可以通过此session id来冒充用户访问应用程序(只要该session id还是有效的，也就是没有被系统重新生成或者销毁)。 通过这种方式，攻击者就不需要捕获用户的Session id(该种方式难度相对稍大)。

当然，以上的解释的前提是攻击者也知道session name了。我们简单讲个概念啊，下面提到的session标识符，php中默认的格式就是PHPSESSID=1234。等号前面的是session name, 后面的是session id。

固定会话ID

在 我看来，Session的安全性应该说是最重要的，也是最复杂的。对于web应用程序来说，加强安全性的第一条原则就是 - 不要信任来自客户端的数据，一定要进行数据验证以及过滤，才能在程序中使用，进而保存到数据层。 然而，为了维持来自同一个用户的不同请求之间的状态， 客户端必须要给服务器端发送一个唯一的身份标识符(Session ID)。 很显然，这和前面提到的安全原则是矛盾的，但是没有办法，http协议是无状态的，为了维持状态，我们别无选择。 可以看出，web应用程序中最脆弱的环节就是

session，因为服务器端是通过来自客户端的一个身份标识来认证用户的，所以session是web应用程序中最需要加强安全性的环节。

基于session的攻击有很多种方式。大部分的手段都是首先通过捕获合法用户的session, 然后冒充该用户来访问系统。也就是说，攻击者至少必须要获取到一个有效的session标识符，用于接下来的身份验证。

据我所知，攻击者至少可以通过以下三种方式来获取一个有效的session标识符:

- 预测
- 捕获（劫持）
- 固定

预 测这种方式，也就是攻击者需要猜测出系统中使用的有效的session标识符，有点类似暴力破解。 php内部session的实现机制虽然不是很安全，但是关于生成session id的关节还是比较安全的，这个随机的session id往往是极其复杂的并且难于被预测出来，所以说，这种攻击方式基本上是不太可能成功的。

捕获一个有效的session标识符 - 这种方式使用的就比较普遍了，很多的攻击类型都是使用这种方式来获取session标识符。当session标识符存储在浏览器的cookie中的时候， 如果浏览器有漏洞的话（比如早期的IE），就就可能暴露其中的session标识符信息。当通过url参数来传递session标识符的话，那就更加容易 暴露这个标识符，有很多方法可以捕获存在于url中的session标识符。所以，一般都是通过cookie来存储传递session标识符，尽管也有可 能因为浏览器的漏洞而被攻击，但是相比通过url来传递，cookie这种方式要安全的多。

Session固定这种方式，简单来讲，攻击者 要想办法，让某个用户通过他预先选择的session标识符来访问系统。一旦系统接收到了这个用户的请求，并且使用用户传递过来的session标识创建 了会话，攻击者就可以使用这个session标识了。这种方式是最简单的获取有效的session标识符，但是

不一定能成功，因为有的系统发现在服务器端 找不到对应该session标识符的数据的话，会自动重新创建一个session标识符，并且保存到客户端。

看个示例

举个最简单的例子，一个可以固定会话id的链接:

<ahref="http://host/index.php?PHPSESSID=1234"> Click here

或者一个php程序中的重导向,也就是应用http协议中的header来进行对请求重导向:

header('Location: http://host/index.php?PHPSESSID=1234');

当 然，关于重导向请求的方法，还有其它几种方式。 比如， 通过HTTP中的Refresh头部来实现以及在html head标签中加入一个具有http-equiv属性的meta tag。不管使用哪种方式，关键点就是要使某个用户访问某个url， 在这个url中包括了攻击者预先设定的session标识符。这就是一般攻击中的第一步，图1对此过程进行了简单描述:

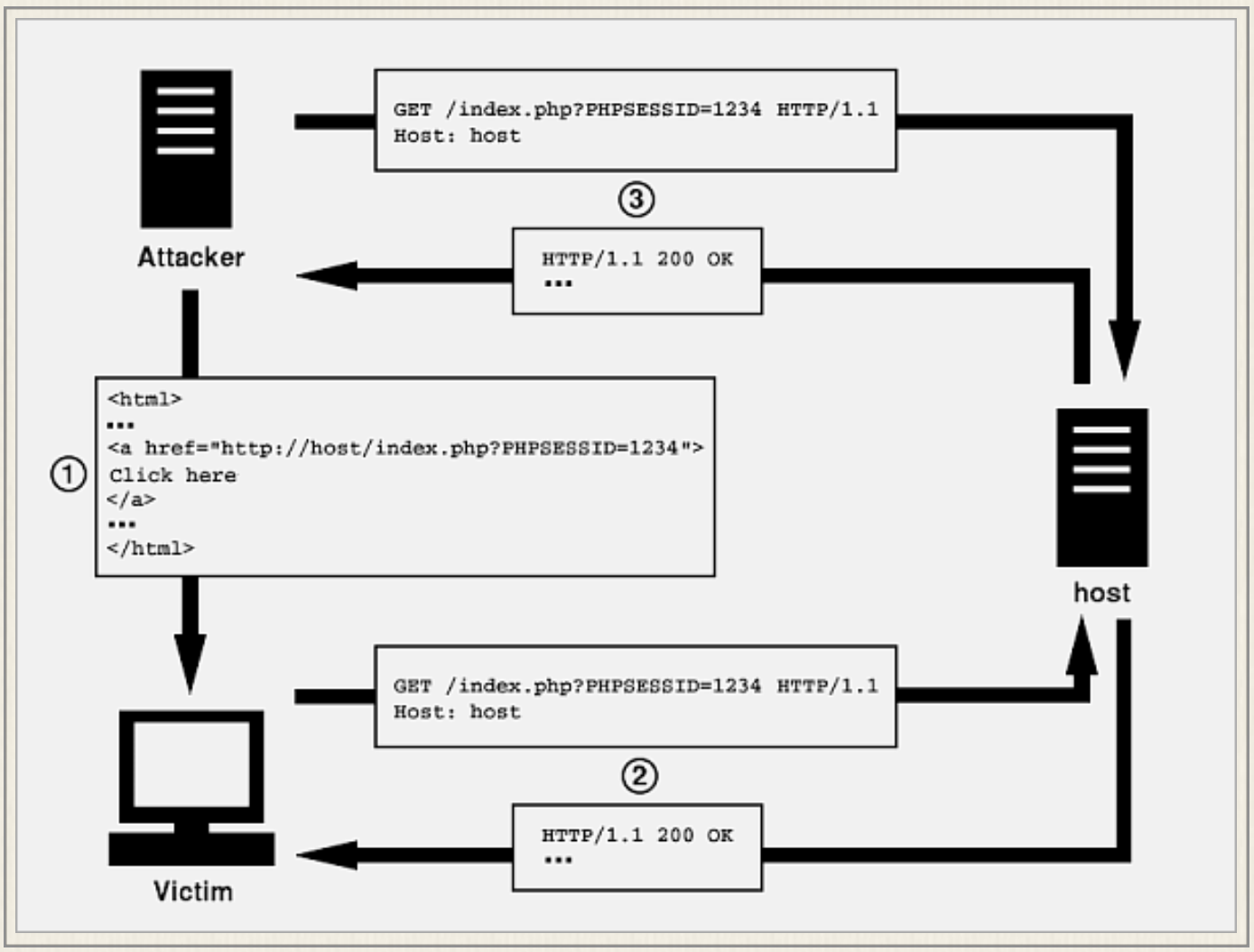


图1. 固定session id示例图

如果系统没有对此进行防御的话，用户的session标识符就这样被攻击者固定住了，接着攻击者就可以通过此session标识符来对系统进行更危险的攻击。

建议你用下面的示例代码1进行一下试验，来验证此种攻击的危险性:

示例代码1

```
session_start();
 if (!isset($_SESSION['count']))
{
$_SESSION['count'] =0;
 }
else
 {
$_SESSION['count']++;
 }
 echo$_SESSION['count'];
```

请 将以上示例代码保存为session.php, 并且放在能通过域名进行测试的目录下面，你可以在本地建立web服务器然后配置一个虚拟域名，这个就不在这里详细讲怎么做了。假设，你在本地可以通过http://www.myhost.me/session.php在firefox浏览器中访问这个文件，请清除firefox浏览器中关于该自定义域 名的所有cookie, 然后再通过http://www.myhost.me/session.php?PHPSESSID=1234再次访问session.php文件。这种 情况下，在第一次访问的时候，这段代码会输出0,刷新页面，将输出1。不断刷新的话，输出的数值会不断增大，这意味着每一次请求的值得到了保留，客户端和 服务端之间的状态得到了保持。

Okay.现在请换用另外一个浏览器进行测试，就用chrome(google的产品确实够快)吧。请同样地访问http://www.myhost.me/session.php?PHPSESSID=1234，你会发现初次访问的输出值不是0，而是在 firefox上

面浏览器中最后输出值基础上增加了1。这说明你已经侵入了前一次创建的session,虽然你在同一台电脑上，但是这两个不同的浏览器就 可以代表两个不同的用户，后者成功冒充成了前者。你甚至可以换一台电脑，只要能访问到http://www.myhost.me/session.php?PHPSESSID=1234,那么你也能在另一台电脑上看到同样的结果。

所以，假如你的系统只是使用了 session_start()， 请尽量不要通过url来传递session标识符，这是相当危险的，因为php默认如果没有发现cookie中的session标识符的话，就会查找是否 有session标识符包含在get或者post数据中，也就是session标识符处于url参数中或者表单的隐藏域中，如果php发现了 session标识符，就不在重新生成一个新的随机session id了，它会使用来自请求中获得的session id来标识此次会话。

当然，这种简单的攻击方式只对那些接受来自url的session标识符的系统起作用，不然就会失败。既然是这样，那么无论在什么情况下，对于第一次请求， 我们必须保证系统要生成一个新的session id,并且颁发给客户端。有很多方法可以实现这个目标，请看下面的示例代码2（这个方式，还是有些问题，不够完善，下面还有更专业些的方案）：

示例代码2

```
session_start();
if (!isset($_SESSION['initiated']))
{
session_regenerate_id();
$_SESSION['initiated'] =true;
 }
```

假如使用上述代码来启动所有的sessions,任何已成功创建的session肯定会包含一个初始的initiated,并且其值为true。如果没有包 含这个初始值，那么可以断定这是个新会话，系统会调用session_regenerate_id这个函数来重新随机生成一个session id，原先的session id会被替代，但是其下的信息还是保留的,如果要删除旧的session的数据，可以给这个函数传递一个bool参数来实现。可以看出，在这种情况下，即 使攻击者已经成功使某个用户

携带预先设定的session id来访问系统，该用户也会被颁发一个新的session id。很明显，这个新的id，攻击者暂时是无法获知的，除非使用其它方式获取。但是，一个老练的攻击者还是有办法来绕过以上检查，请看下节。

老练的攻击

一个经验丰富一些的攻击者，往往会自己先访问目标系统，从而从系统处获得一个合法的session标识符，并且保持这 个session标识符不过期，然后再使用前面讲过的方式，来使某个目标用户使用攻击者获取的，当前有效的session标识符来访问目标系统。这种情况 下，如果该用户登录后（假设在用户登录时，系统没有重新生成session id），那么攻击者就可以通过同样的session标识符来侵入用户在该系统中的账务，这是很危险的事情，意味着用户的敏感信息会被泄露，或者导致后续更 危险的操作的发生。所以，在用户登录的时候，切记一定要重新生成session id,因为这个时刻关于此用户的敏感信息会被保存到session数据中。建议你在认证了用户的用户名以及密码后，可以先调用 session_regenerate_id来重新生成一个session id,然后再初始化一个表明用户处在登录状态的session变量，例如下面的示例代码：

```
session_regenerate_id();

$_SESSION['logged_in'] =true;
```

以 上代码，可以有效地抵御固定会话ID的攻击。事实上，最好在用户权限发生变化的时候或者因为闲置时间太长而导致session过期，用户再次进行验证身份 的时候，就进行一次session id的重新生成。这样的话，可以肯定的说，你的系统就不会受到以上攻击方式的侵害。

由此可见，以上方式比前面的例子安全级别高多了，因为我们给攻击者制造了另外一层障碍，这道障碍有效地防止了攻击者先获取合法session标识符，并且保持有效性，然后再实施侵入。

总结

总的来说，抵御固定会话ID此类攻击的最有效的措施就是在用户提供验证信息，相应的权限级别发生改变的时候，就进行session id的重新生成。当然，情况总是不断变化，这不总是万无一失的。

原文链接：http://blogread.cn/it/article/6682?f=hot1

WWDC 2014 Session笔记 - 可视化开发，IB 的新时代

作者：王巍

WWDC 2014 Session笔记 - 可视化开发，IB 的新时代

- What's New in Xcode 6

http://devstreaming.apple.com/videos/wwdc/2014/401xxfkzfrjyb93/401/401_whats_new_in_xcode_6.pdf?dl=1

- What's New in Interface Builder

http://devstreaming.apple.com/videos/wwdc/2014/411xx0xo98zzoor/411/411_whats_new_in_interface_builder.pdf?dl=1

如果说在 WWDC 14 之前 Interface Builder (IB) 还是可选项的话，我相信在此之后 IB 已经是毫无疑问的 iOS 开发标配了，纯代码界面可以说已经渐行渐远，可以逐渐离开我们的视线了。

一言蔽之，就是 Apple 在催促大家使用 IB，特别是 Storyboard 做为界面开发的唯一选择这件事上，下定了决心，也做出了实际的行动。

如果是纯代码 UI 在此之前还能有所挣扎的话，那么压死这个方案的最后一根稻草就是 Size Classes。我已经在之前的笔记中 对这方面内容做了些简单的探索，但是还远远不够，也许在将来某一天我还会重新整理下 Size Classes 这个主题的内容，以及使用 IB 适配不同屏幕的一些实践，但是不是这次。这篇文章里想要介绍的是 Xcode 6 中为 IB 锦上添花的一个特性，那就是实时地预览自定义 view，这个特性让 IB 开发的流程更加直观可视，也可以减少很多无聊的参数配置和 UI 设置的时间。

以前 IB 的不足

作为可视化开发的工具，IB 和 Storyboard 在组织和构建 ViewController 及其导航关系时已经做得很好的。对于 ViewController 的 view 画布上的诸如 UILabel 或者 UIImageView 这样的基础的类，IB 是能够很好地支持并实时在设计的时候进行显示的。但是对于那些自定义的类，之前的 IB 就束手无策了。我们能做的仅仅是在 IB 中拖放一个 UIView，然后通过将 Custom Class 属性设置为我们自定义的 UIView 的子类来在 “暗示” IB 在运行时初始化一个对应的子类。这样的问题是在开发自定义的 view 时，我们不得不一遍遍地修改代码并运行，再根据运行结果进行调整和修正。而实际上，单一对某个 view 的调试这种问题只涉及到设计层面，而非运行层面，如果我们能够在设计时就有一个实时地对自定义 view 的预览该多好。

没错，Apple 也是这么想的，并且在 Xcode 6 中，我们就已经可以创建这样的 UIView 子类了：利用新加入的 @IBDesignable 和 @IBInspectable，我们可以非常方便地完成在 IB 中实时显示自定义视图，甚至和其他一些内置 UIView 子类一样，直接在 IB 的 Inspector 改变某些属性，甚至我们还能通过设置断点来在 IB 中显示视图时进行调试。新的这些特性非常强大，使用起来却出乎意料的简单。下面我将通过一个实际的小例子加以说明。最终的完整例子已经放在 GitHub 上了，现在我们从开始一步步开始吧。这些代码基于 Xcode 6.1 和 Swift 1.1。

时钟 view 的例子

单纯的自定义 view

假设我们有一个自定义的 view，用来描画一个时钟，如果有在读 objc.io 或者 objc 中国 的读者，可能会发现这段代码是动画一章一篇文章里代码的改造过的 Swift 版本。

在这里我们有一个自定义的 UIView 的子类：ClockFaceView，其中嵌套了一个 ClockFaceLayer 作为 layer。如果我们不需要动画，我们也可以简单地使用 -drawRect: 来完成绘制。但是在这里我们还是选择使用添加 CALayer 的方式，这会使之后做动

画简单好几个数量级 -- 因为我们可以简单地通过 CA 动画而不是每帧去计算绘制来完成动画 (在这篇帖子里不会涉及这些内容)。

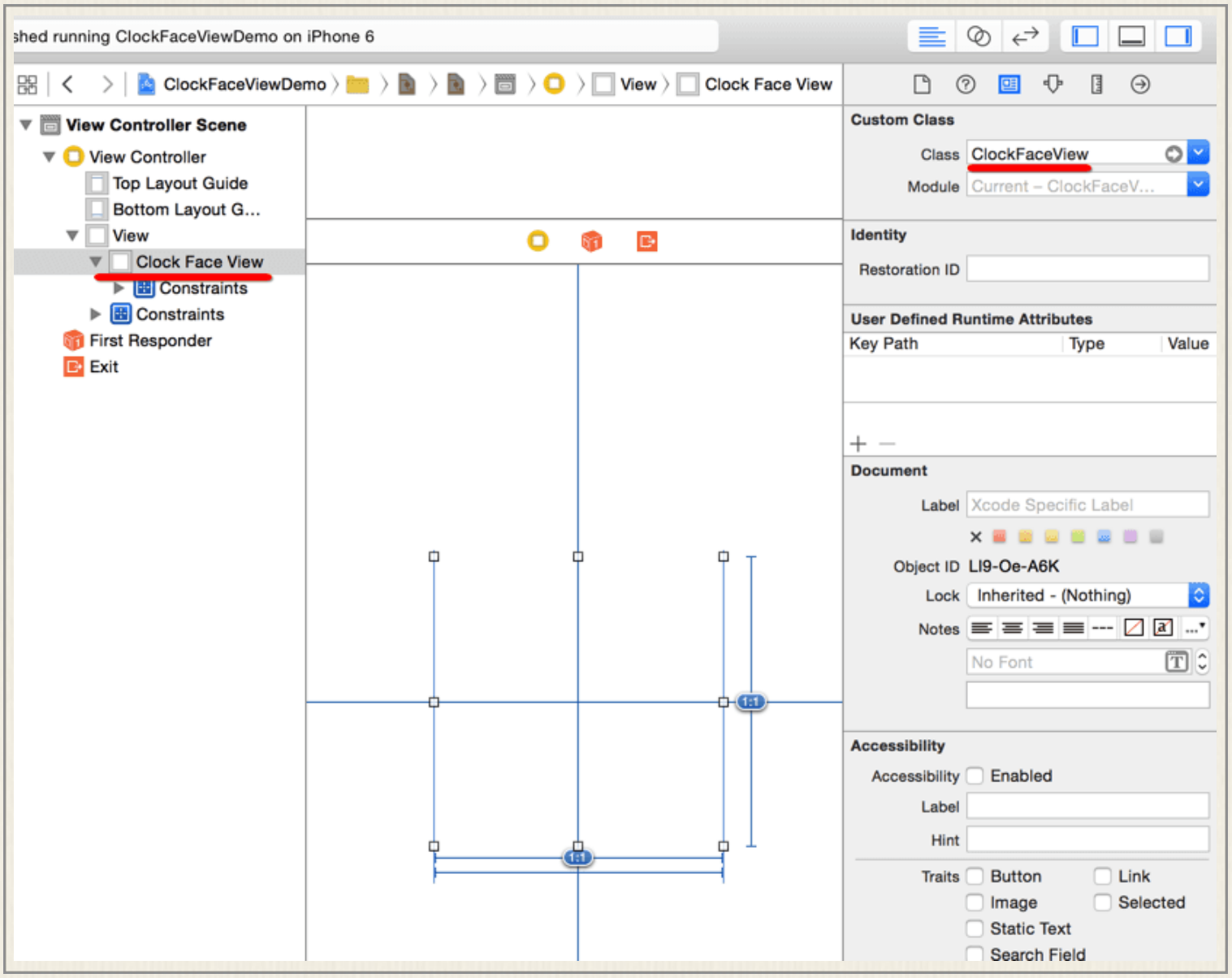
```
// ClockFaceView.swift
import UIKit

class ClockFaceView : UIView {

    class ClockFaceLayer : CAShapeLayer {

        private let hourHand: CAShapeLayer
        private let minuteHand: CAShapeLayer

        override init() {
            hourHand = CAShapeLayer()
            minuteHand = CAShapeLayer()

            super.init()
            frame = CGRect(x: 0, y: 0, width: 200, height: 200)
            path = UIBezierPath(ovalInRect: CGRectInset(frame, 5,
5)).CGPath
            fillColor = UIColor.whiteColor().CGColor
            strokeColor = UIColor.blackColor().CGColor
            lineWidth = 4
```

```
        hourHand.path = UIBezierPath(rect: CGRect(x: -2, y: -70, width:
4, height: 70)).CGPath

        hourHand.fillColor = UIColor.blackColor().CGColor

        hourHand.position = CGPoint(x: bounds.size.width / 2, y:
bounds.size.height / 2)

        addSublayer(hourHand)


        minuteHand.path = UIBezierPath(rect: CGRect(x: -1, y: -90,
width: 2, height: 90)).CGPath

        minuteHand.fillColor = UIColor.blackColor().CGColor

        minuteHand.position = CGPoint(x: bounds.size.width / 2, y:
bounds.size.height / 2)

        addSublayer(minuteHand)
    }

    required init(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    func refreshToHour(hour: Int, minute: Int) {

hourHand.setAffineTransform(CGAffineTransformMakeRotation(CGFloat(
Double(hour) / 12.0 * 2.0 * M_PI)))
```

```
minuteHand.setAffineTransform(CGAffineTransformMakeRotation(CGFloa
t(Double(minute) / 60.0 * 2.0 * M_PI)))
        }
    }

    private let clockFace: ClockFaceLayer

    var time: NSDate? {
        didSet {
            refreshTime()
        }
    }

    private func refreshTime() {
        if let realTime = time {
            if let calendar = NSCalendar(calendarIdentifier: NSGregorianCal-
endar) {
                let components =
calendar.components(NSCalendarUnit.CalendarUnitHour |
                                        NSCalendarUnit.CalendarUnitMinute,
fromDate: realTime)
                clockFace.refreshToHour(components.hour, minute:
components.minute)
            }
        }
```

```
}

override init(frame: CGRect) {
    clockFace = ClockFaceLayer()
    super.init(frame: frame)
    layer.addSublayer(clockFace)
}

required init(coder aDecoder: NSCoder) {
    clockFace = ClockFaceLayer()
    super.init(coder: aDecoder)
    layer.addSublayer(clockFace)
  }
}
```

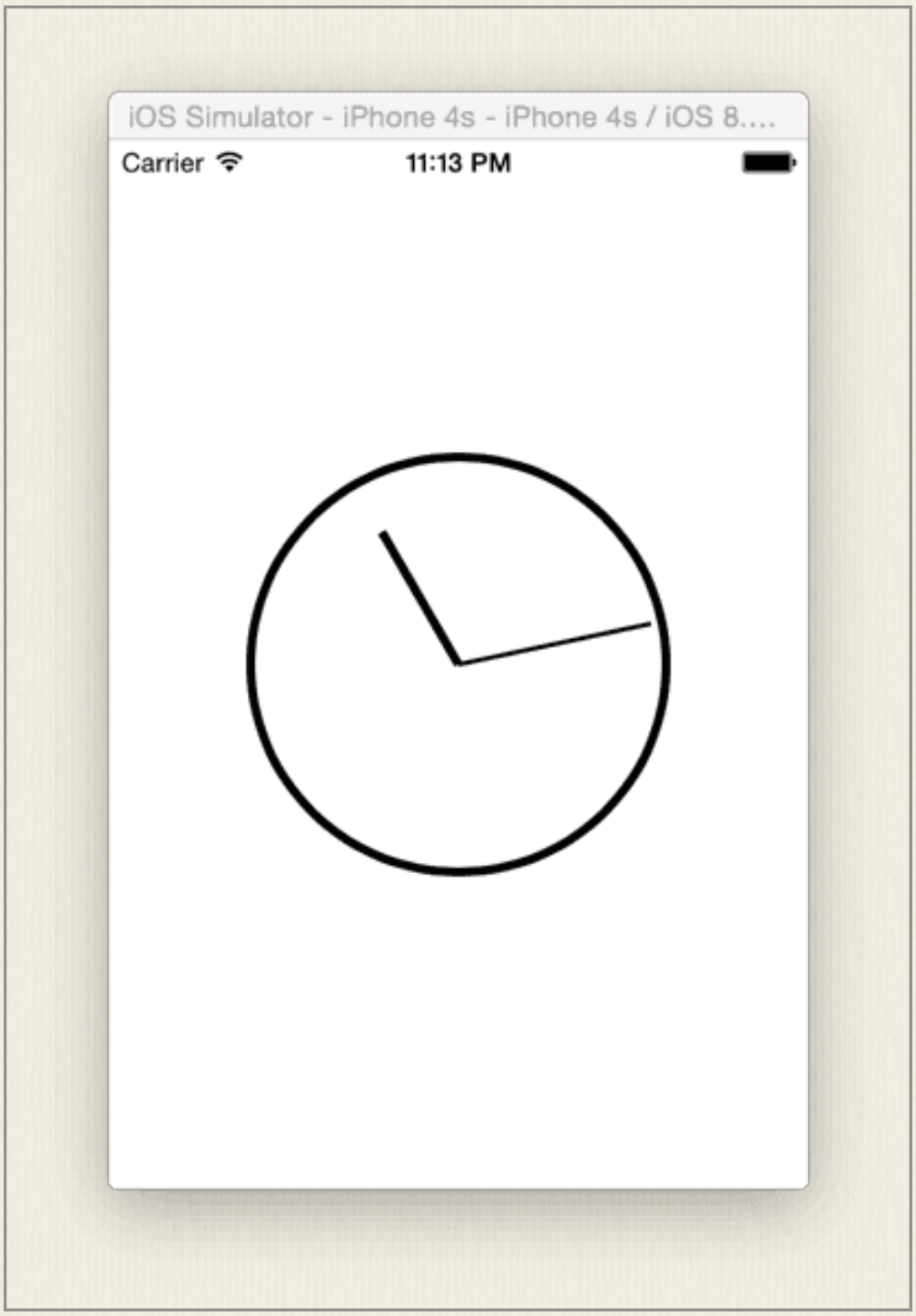
如果你没有耐心看完的话也没有关系，简单来说就是 ClockFaceView 在被初始化时会向自己添加一个 ClockFaceLayer，用来显示分针和时针。通过设置 time 属性我们可以更新时钟的位置。因为提供了 initWithCoder:，因此我们是可以直接从 IB 里加载这个 view 的。方法就是最普通的类型指定，并让 app 在加载时初始化对应的类型：在新建的 Single View Application 的 Storyboard 中添加一个 UIView 控件，然后设置好约束，并且将 Class 设置为 ClockFaceView：



运行应用，可以看到 ClockFaceView 被正确地初始化了，指针指向默认的 12 点整。通过为这个 view 建立 outlet 或者用其他 (比如 tag 的方式，虽然我不太喜欢这么做，但是我见过不少人这么弄) 方法找到这个 ClockFace-View 并设置时间的话，我们可以正确地改变其时针和分针的指向：

```
// ViewController.swift
class ViewController: UIViewController {

    @IBOutlet weak var clockFaceView: ClockFaceView!

    override func viewDidLoad() {
        super.viewDidLoad()
```

```
        // Do any additional setup after loading the view, typically from a
nib.

        clockFaceView.time = NSDate()
    }
}
```

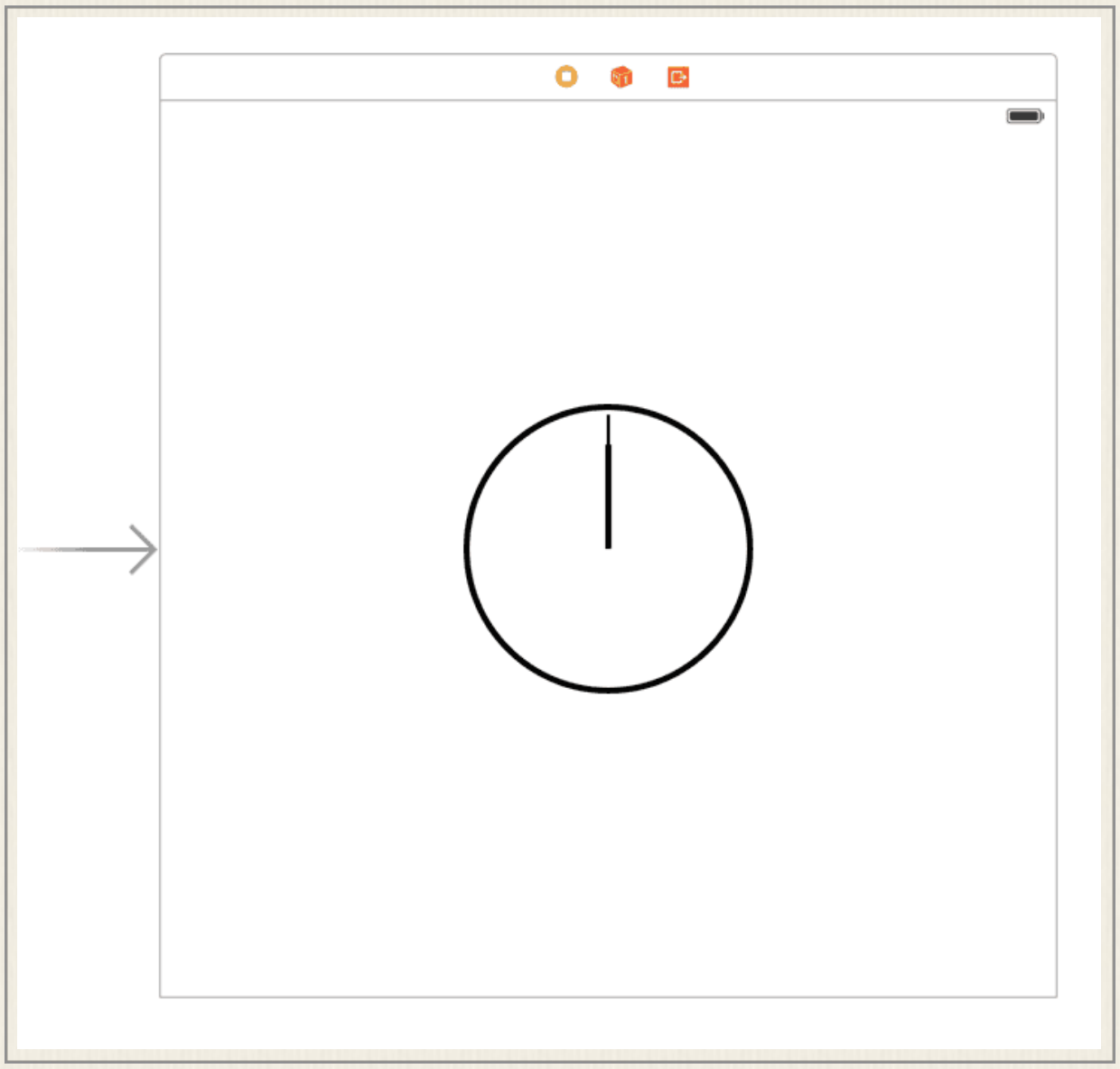


IBDesignable，IB 中自定义 view 的渲染

把大象装进冰箱有三个步骤，而让 IB 显示自定义 view 居然只有一个步骤！

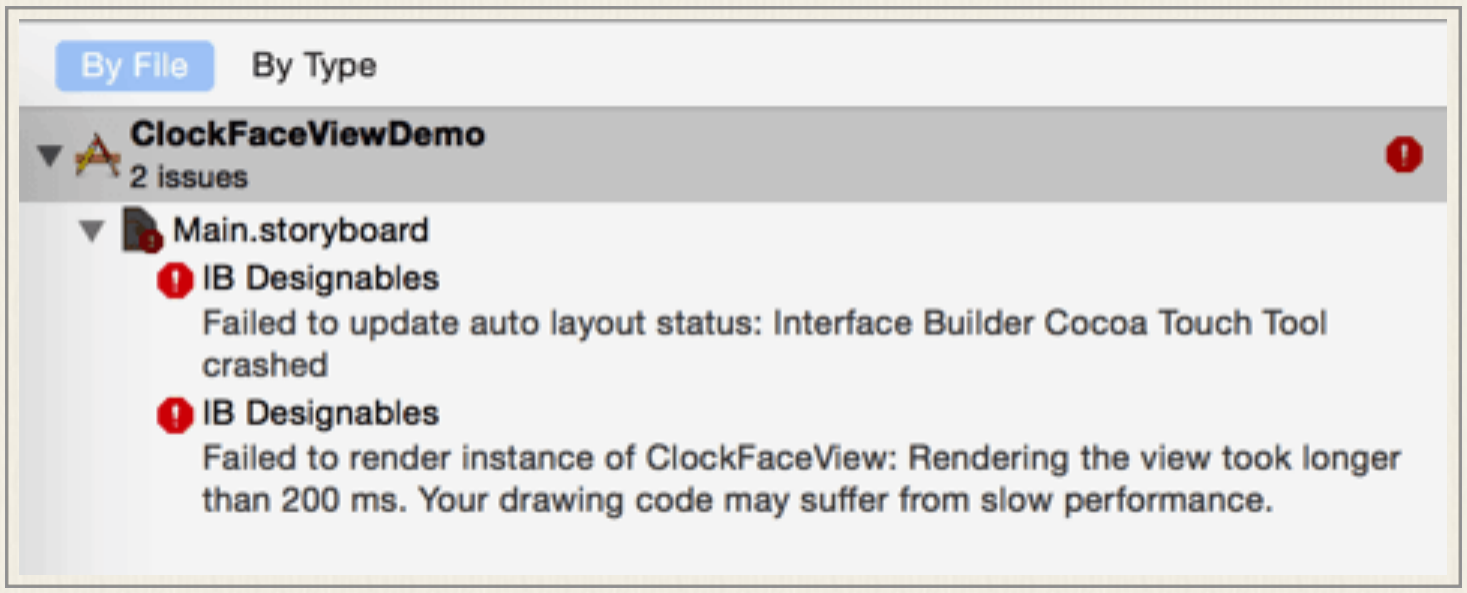
只要我们在 class ClockFaceView : UIView 这个类型定义上面加上一个 @IBDesignable 的标记，就完成了！

在进行更改并等待编译和 IB 自动识别后，我们就可以在 IB 中原来一块白色的地方看到初始化后的时钟了：



如你所想，这个标记的作用是告诉 IB 如果遇到对应的 UIView 子类的话，可以对其进行渲染。深入一些来说，IB 将寻找你的子类中的 -initWith-Frame: 方法，并给入当前自定义 view 的 frame 对其进行调用。需要注意的是，在使用 IB 初始化 view 时，被调用的是 -initWithCoder: 而非 frame 版本，所以说在想要实现自定义 view 在 IB 中的预览的话，我们至少必须

实现这两个版本的初始化方法。不过好消息是，如果我们只添加了 @IB-Designable，而忘了实现 -initWithFrame: 的话，在 IB 渲染 view 时会给我们抛出大大的错误，所以因为遗漏而花大量时间在查找哪里出了问题这种事情应该不太可能发生。



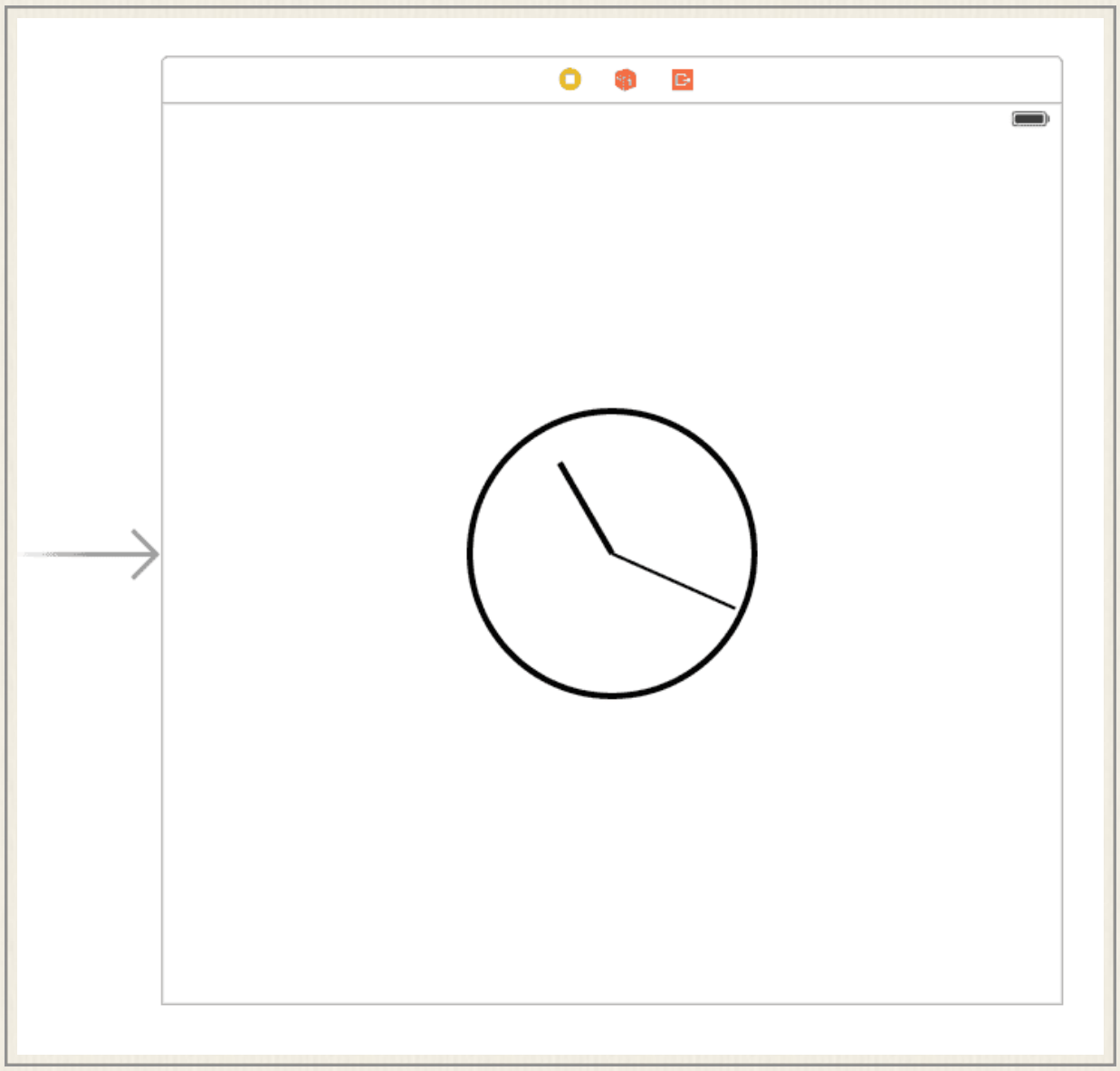
仅设计时的配置

现在在 IB 中我们显示的时钟只能默认地指向 0 点 0 分，这是因为在设计的时候，我们并没有机会去设定这个 view 的 time 属性，所以时针和分针都停留在了初始的位置上。在 Xcode 6 中可以在 @IBDesignable 标记的 UIView 子类中添加一个 prepareForInterfaceBuilder 方法。每次在 IB 即将把这个自定义的 view 渲染到画布之前会调用这个方法进行最后的配置。比如我们想在 IB 中这个时钟的 view 上显示当前时间的话，可以在 Clock-FaceView 中加入这个方法：

```
class ClockFaceView : UIView {
    //...

    override func prepareForInterfaceBuilder() {
        time = NSDate()
    }
```

```
    //...
}
```

保存并切换到 IB，静待自动编译和执行，可以看到类似下面的结果：



挺好的...现在我们的 IB 不仅被用来设计界面了，还兼备了看时间的功能 - 虽然这个时钟并不是实时的，只有在切换编辑器界面到 IB 或者是修改了相关文件时才会进行刷新。

另外虽然这篇文章没有涉及，但是需要一提的是，如果你想要在 prepareForInterfaceBuilder 里加载图片的话，需要弄清楚 bundle 的概念。IB 使用的 bundle 和 app 运行时的 mainBundle 不是一个概念，我们需要在设计时的 IB 的 bundle 可以通过在自定义的 view 自身的 bundle 中进行查找并使用。比如想要加载一张名为 image.png 的图片的话：

```
let bundle = NSBundle(forClass: self.dynamicType)
if let fileName = bundle.pathForResource("image", ofType: "png") {
    if let image = UIImage(contentsOfFile: fileName) {
        // 在此处可以使用 image
    }
}
```

在使用 IB 中的方法读取资源时一定要注意运行环境不同这一点。

用 IBInspectable 在 IB 中调整属性

IBDesignable 的 view 的另一个很方便的地方是我们可以向 Inspector 中添加自定义的内容了。通过这样做，就可以直接在 IB 中对 view 进行一些编辑和配置。以前对于自定义 view，我们通常只能通过用类似 IBOutlet 的方式在代码中进行设置，或者是配置 Runtime Attribute 来进行，而现在我们有能力直接通过像给一个 UILabel 设定字符串或者给 UIImageView 设定图片这样的方式来设置自定义 view 的部分属性了，这也使得在 IB 中的自定义 view 的易用性和完整性得到了极大增强。

使用方法也非常简单，只需要在某个属性前加上 @IBInspectable 标记即可。比如我们可以在 ClockFaceView 中加入以下代码：

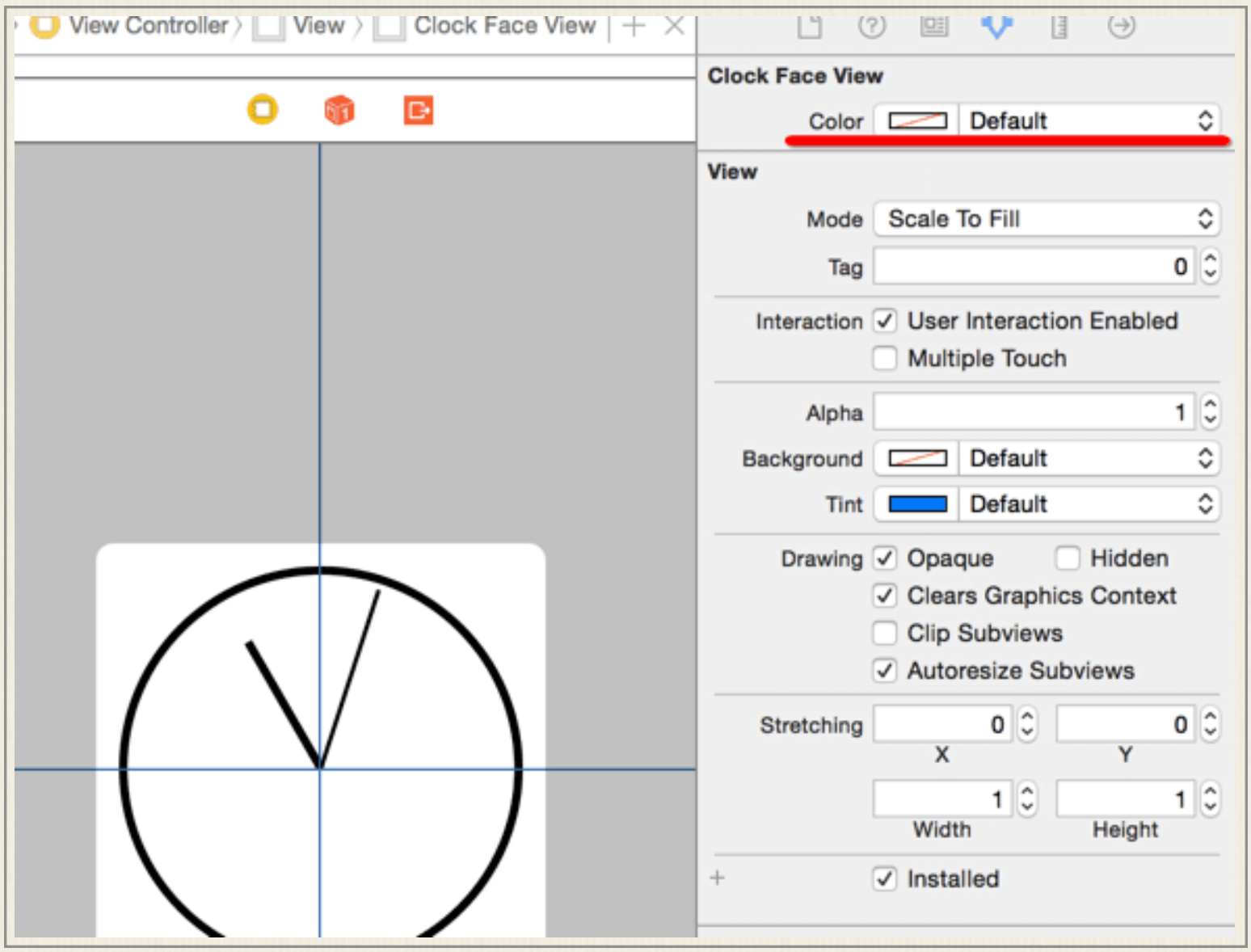
```
class ClockFaceView : UIView {
    //...

    @IBInspectable
    var color: UIColor? {
        didSet {
            refreshColor()
        }
```

```
    }

    private func refreshColor() {
        if let realColor = color {
            clockFace.refreshColor(realColor)
        }
    }

    //...
}
```

然后在 *ClockFaceLayer* 中加入对应的 *refreshColor* 方法：

```
class ClockFaceLayer : CAShapeLayer {
    //...

    func refreshColor(color: UIColor) {
        hourHand.fillColor = color.CGColor
        minuteHand.fillColor = color.CGColor
        strokeColor = color.CGColor
    }

    //...
}
```

我们对 ClockFaceView 中的 color 属性添加了 @IBInspectable，在保存和编译后，这会在 IB 中对应的 view 的 Attribute Inspector 中添加一个颜色选取的属性：



当我们在 IB 中设置这个属性的时候，对应的 didSet 将会被执行，通过 refreshColor 方法就可以直接改变 IB 中这个 view 的时针和分针的颜色了。

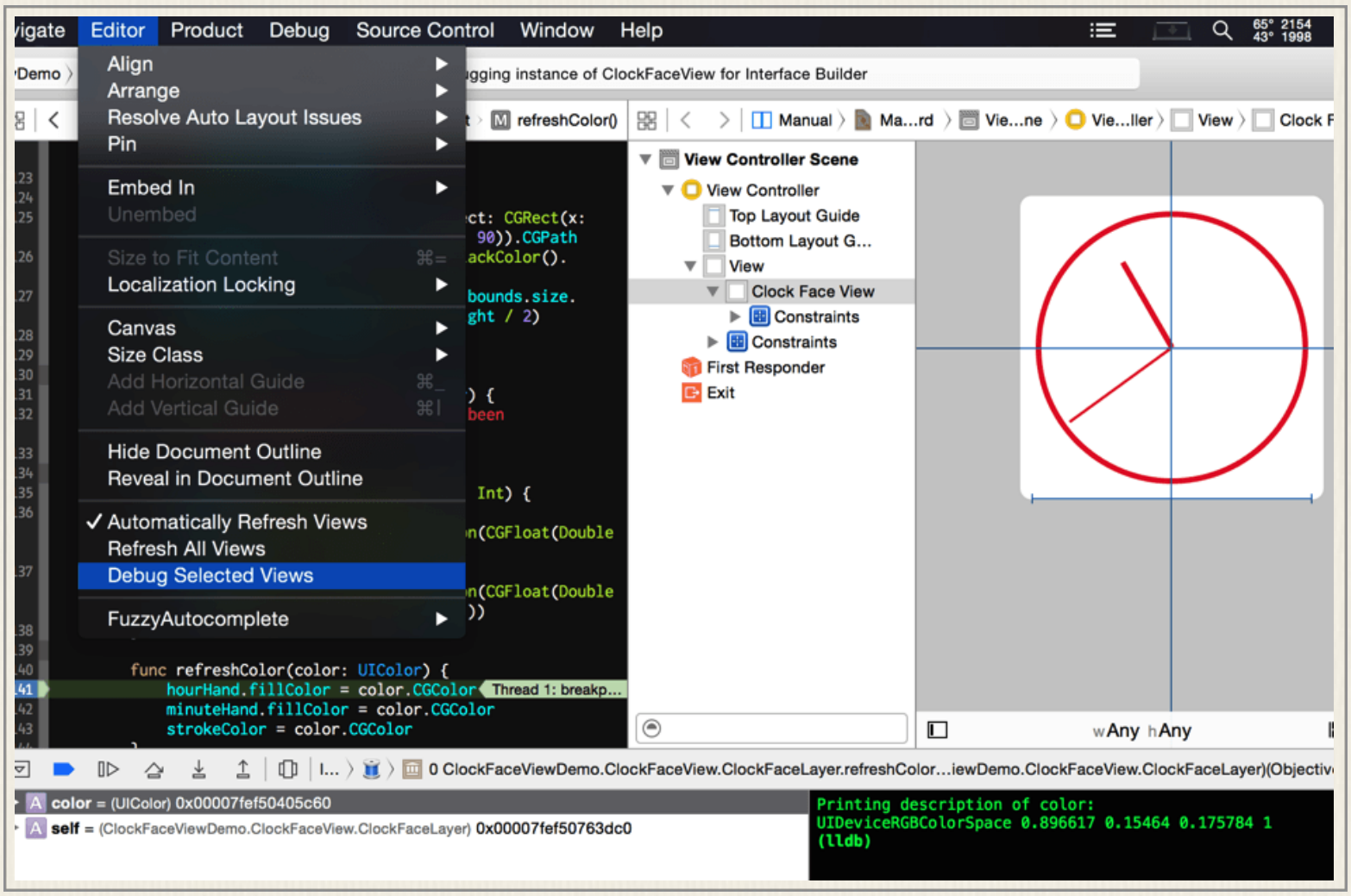
注意这个改变并不像 prepareForInterfaceBuilder 那样仅发生在设计时，我们直接运行代码，会看到运行时的颜色也是发生了改变的。其实 @IBInspectable 并没有做什么太神奇的事情，我们如果查看 IB 中这个 view 的 Identity Inspector 的话会看到刚才所设定的颜色值被作为了 Runtime Attribute 被使用了。其实手动直接在 Runtime Attributes 中设定颜色也有同样的效果，因此 @IBInspectable 唯一做的事情就是在 IB 面板上为我们提供了一个很方便地修改属性的入口，别没有其他太多神奇之处。

这个原理同时也决定了 @IBInspectable 是有一定限制的，即只有能够在 Runtime Attributes 中指定的类型才能够被标记后显示在 IB 中，这些类型包括 Boolean，Number，String，Ponit，Size，Rect，Range，Color 和 Image。像是如果想要把类似 time 这样的属性标记为 @IBInspectable 的话，

在 IB 中还是无法显示的，因为 Xcode 并没有准备 NSDate 类型。不过其实通过 KVC 进行动态设定这种事情在原理上是没有问题的，界面的支持应该也可以通过 Xcode 插件进行扩展，感觉上并不是一件特别困难的事情，有兴趣的同学不妨尝试，应该挺有意思 (当然也有可能会是个坑)。

自定义渲染 view 的调试

对于简单的自定义 view 来说，实时显示和属性设定什么的并不是一件很难的事情。但是对于那些比较复杂的 view，如果我们遇到某些渲染上的问题的话，如果只能靠猜的话，就未免太可怜了。幸好，Apple 为 view 在 IB 中的渲染的调试也提供了相应的方法。在 view 的源代码中设置好断点，然后切到 IB，点选中我们的自定义的 view 后，我们就可以使用菜单里的 Editor -> Debug Selected Views 来让 IB 对这个自定义 view 进行渲染。如果触发了代码中的断点，那我们的代码就会被暂停在断点处，lldb 也会就位听我们调遣。一切都感觉良好，不是么？



总结

Xcode 6 中的很多 key feature 都是基于或者重度依赖 Interface Builder 的。比如 Size Classes，比如 xib 的启动画面，再比如本篇文章中说到的自定义 view 渲染等等。在 iOS 或者 Mac 开发中，IB 现在处于一个比以往任何时候都重要的时期，使用 IB 和这些方便的特性进行开发已经从可选项变为了必须项。很难想象没有 IB 的话要怎么才能使用这些工具，更进一步地说，很难想象没有 IB 的话开发者需要浪费多少时间在本应该迅速完成的工作中。

如果你还在使用代码来构建 UI 的话，现在也许是你最后的放下代码，拿起 IB 武装自己的机会了。一开始可能会有迷惑，会不习惯，会觉着被拽出了舒适区浑身无力。但是一旦适应以后，你不仅能够收获最新的技能和工具，也有机会站在一个 全新的高度，来审视 app 中界面开发的种种，并从中找到乐趣。

P.S. 如果你不知道要从哪里入手，推荐可以从 raywenderlich 家的这篇 AutoLayout 教程开始你的 IB 之旅。

原文链接：http://onevcat.com/2014/10/ib-customize-view/

iOS工程如何支持64-bit

作者：Chun Tips

苹果在2014年10月20号发布了一条消息：从明年的二月一号开始，提交到App Store的应用必须支持64-bit。详细消息地址为：https://developer.apple.com/news/?id=10202014a

那们我们应该如何开始着手让自己的App支持64-Bit呢？

基本知识

从iPhone 5S的A7 CPU开始到刚刚发布的iPhone 6（A8 CPU）都已经支持64-bit ARM 架构。关于64-bit 的介绍详见维基百科。知乎上有很多关于苹果使用A7，A8芯片的讨论，可以参考 iPhone 6 的 Apple A8 芯片对比 Apple A7 提升明显吗？, iPhone 5s 配备的 A7 处理器是 64 位，意味着什么？（http://www.zhihu.com/question/25285088 ）

- Xcode 5.0.1开始支持编译32-bit和64-bit的Binary
- 同时支持32-bit和64-bit，我们需要选择的minimum deployment target为 iOS 5.1.1
- 64-bit的Binary必须运行在支持64-bit 的CPU上，并且最小的OS版本要求是 7.0.3

关于Xcode “Build Setting”中的Architectures参数问题

- Architectures：你想支持的指令集。（支持指令集是通过编译生成对应的二进制数据包实现的，如果支持的指令集数目有多个，就会编译出包含多个指令集代码的数据包，造成最终编译的包很大。）
- Valid architectures：即将编译的指令集。（Valid architectures 和 Architecture两个集合的交集为最终编译生成的版本）

- Build Active Architecture Only：是否只编译当前设备适用的指令集（如果这个参数设为YES，使用iPhone 6调试，那么最终生成的一个支持ARM64指令集的Binary。一般在DEBUG模式下设为YES，RELEASE设为NO）

关于指令集如下参考：

ARMv8/ARM64: iPhone 6(Plus), iPhone 5s, iPad Air(2), Retina iPad Mini(2,3)

ARMv7s: iPhone 5, iPhone 5c, iPad 4

ARMv7: iPhone 3GS, iPhone 4, iPhone 4S, iPod 3G/4G/5G, iPad, iPad 2, iPad 3, iPad Mini

ARMv6: iPhone, iPhone 3G, iPod 1G/2G

对于支持64-bit,我们可以设置Architectures为 Standard architectures，在最新的Xcode 6上，它包括 armv7和arm64。

让App支持32-bit和64-bit基本步骤

- 确保Xcode版本号>=5.0.1
- 更新project settings, minimum deployment target >= 5.1.1
- 改变Architectures为 Standard architectures（include 64-bit）
- 运行测试代码，解决编译warnings and errors，对照本文档或者官方文档 64-Bit Transition Guide for Cocoa Touch对相应地方做出修改。（编译器不能告诉我们一切）
- 在真实的64-bit机器上测试
- 使用Instruments查看内存使用问题

64-bit主要的变化

64-bit运行时环境和32-bit运行时环境主要有以下两点的不同:

- 数据类型的改变
- 方法调用上的改变

数据类型的改变

整型数据类型的变化如下:

整型数据类型	32-bit Size	32-bit Alignment	64-bit Size	64-bit Alignment
char	1 byte	1 byte	1 byte	1 byte
BOOL, bool	1 byte	1 byte	1 byte	1 byte
short	2 byte	2 byte	2 byte	2 byte
int	4 byte	4 byte	4 byte	4 byte
long	4 byte	4 byte	**8 byte**	**8 byte**
long long	8 byte	4 byte	8 byte	**8 byte**
pointer	4 byte	4 byte	**8 byte**	**8 byte**
size_t	4 byte	4 byte	**8 byte**	**8 byte**
time_t	4 byte	4 byte	**8 byte**	**8 byte**
NSInteger	4 byte	4 byte	**8 byte**	**8 byte**
CFIndex	4 byte	4 byte	**8 byte**	**8 byte**
fpos_t	8 byte	4 byte	8 byte	**8 byte**
off_t	8 byte	4 byte	8 byte	**8 byte**

关于字节对齐的概念可以参考如下链接: http://blog.csdn.net/21aspnet/article/details/6729724#comments

浮点型类型的改变如下:

浮点型数据类型	32-bit Size	64-bit Size
float	4 byte	4 byte
double	8 byte	8 byte
CGFloat	4 byte	**8 byte**

数据类型的改变可能会为我们的程序带来这些影响:

- 增加内存压力
- 64-bit到32-bit数据之间的相互转化
- 计算可能产生不同的结果
- 当把一个值从大的数据类型拷贝到小的数据类型，数据可能被截断。（NSInteger -> int）

方法调用上的改变

基于32-bit的CPU和基于64-bit上的CPU有不同数量的寄存器，在方法调用上有不同的协议。因此32-bit和64-bit在汇编层级上是不同的。如果我们在程序中不使用汇编编程，调用协议很少会遇到。

如何编写健壮的64-bit代码

根据上述改变，官方文档 64-Bit Transition Guide for Cocoa Touch给出如下7步:

- 不要将长整型long赋值给整型int (64-bit上会导致数据丢失)
- 不要将指针类型pointer赋值给整型int (64-bit导致地址数据丢失)
- 留意数值计算(掩码计算,无符号整数和有符号整数同时使用等)

- 留意对齐方法带来的变化
- 32-bit到64-bit 之间数据转化(通过网络传递的用户数据，可能同时存在于32-bit和64-bit的环境下)
- 重写汇编代码
- 不要在可变参数方法和不可变参数方法之前进行强制转化

在LLVM编译器中，枚举类型也可以定义枚举的大小。我们在使用中，指派枚举值到一个变量时，应该使用适当的数据类型。

不要将指针类型pointer赋值给整型int

```
int a = 5;
int *c = &a;

/* 32-bit下正常，64-bit下错误。最新的Xcode6.0编译提示警告:'Cast to int* for smaller integer type int'*/
int *d = (int *)((int)c + 4);

/* 正确, 指针可以直接增加*/
int *d = c + 1;
```

如果我们一定要把指针转化为整型，可以把上述代码改为:

```
/* 32-bit和64-bit都正常。*/
int *d = (int *)((uintptr_t)c + 4);
```

查看uintptr_t定义为 typedef unsigned long uintptr_t;

保持数据类型一致

方法使用时，入参，出参和赋值都需要注意保持数据类型一致。在iOS App中尤其要注意以下几个类型的正确使用：

- long
- NSInteger
- CFIndex
- size_t

在32-bit和64-bit下，fpos_t和off_t都是64 bits的数据大小，永远不要把它们指向int整型。

```
long PerformCalculation(void);
int c = PerformCalculation(); // 错误 64-bit上数据将被截取
long y = PerformCalculation(); // 正确
```

```
int PerformAnotherCalculation(int input);
long i = LONG_MAX;
int x = PerformCalculation(i); // 错误

int ReturnMax()
{
    return LONG_MAX; // 错误
}
```

Cocoa中常见的数据类型转化问题

NSInteger : 在32-bit和64-bit下有分别的定义:

#if __LP64__ || (TARGET_OS_EMBEDDED && !TARGET_OS_I-PHONE) || TARGET_OS_WIN32 || NS_BUILD_32_LIKE_64

typedef long NSInteger;

#else

typedef int NSInteger;

#endif

我们永远不应该假设NSInteger 和int是一样大的，下面的例子在使用中就需要注意:

- 使用NSNumber对象转化时
- 使用NSCoder编解码的时候，如果在64-bit设备下对NSInteger编码，在32-bit设备下对NSInteger解码。解码时如果值的大小超过了32-bit，这个时候就会出现异常
- Famework中使用NSInteger定义的一些常量

CGFloat: 和NSInteger一样有不同的定义

typedef CGFLOAT_TYPE CGFloat;

#if defined(__LP64__) && __LP64__

 # define CGFLOAT_TYPE double

#else

 # define CGFLOAT_TYPE float

#endif

下面给出错误示范:

CGFloat value = 200.0;

*CFNumberCreate(kCFAllocatorDefault, kCFNumberFloatType, &value); //64-bit*下出现错误

CGFloat value = 200.0;

*CFNumberCreate(kCFAllocatorDefault, kCFNumberCGFloatType, &value); //*正确

整型数值计算问题

关于C语言的符号位扩展可参考资料为：http://blog.163.com/shi_shun/blog/static/237078492010651063936/

我们直接来看例子:

int a = -2;

unsigned int b = 1;

long c = a + b;

long long d = c;

printf("%lld\n", d);

问题：这段代码在32-bit下运行结果符合我们的预期，输出为-1(0xffffffff)。在64-bit下运行结果为：4294967295 (0x00000000ffffffff)。

原因：一个有符号的值和一个同样精度的无符号的值相加结果是无符号的。这个无符号的结果被转换到更高精度的数值上时采用零扩展。

解决方案：把变量b换成长整型long

创建数据结构时使用合适的数据大小

C99提供了内置的数据类型保证了一致的数据大小，即使底层的硬件结构不同。在某些case下，我们知道数据是一个固定的大小或者一个特定的变量拥有一个有限的取值范围。这个时候，我们应该选择特定的类型以避免浪费内存。

类型如下：

类型	取值范围
int8_t	-128 to 127
int16_t	-32,768 to 32,767
int32_t	-2,147,483,648 to 2,147,483,647
int64_t	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
uint8_t	0 to 255
uint16_t	0 to 65,535
uint32_t	0 to 4,294,967,295
uint64_t	0 to 18,446,744,073,709,551,615

永远不要使用malloc去为变量申请特定内存的大小，改为使用sizeof来获取变量或者结构体的大小。

另外我们还需要注意修改格式化字符串来同时支持32-bit和64-bit。

类型	Format String
int	%d
long	%ld
long long	%lld
size_t	%zu
ptrdiff_t	%td
any pointer	%p

小心处理方法和方法指针

```
int fixedFunction(int a, int b);

int variadicFunction(int a, ...);

int main
{
    int value2 = fixedFunction(5,5);
    int value1 = variadicFunction(5,5);
}
```

上述两个方法中，在32-bit 下使用相同的指令读取参数的数据，但是在64-bit上，是使用完全不同的协议来编译的。

如果在代码中传递方法指针，应该保证方法调用的协议是一致的。永远不要将一个可变参数的方法转化成固定参数的方法。

```
int MyFunction(int a, int b, ...);

int (*action)(int, int, int) = (int (*)(int, int, int)) MyFunction;
action(1,2,3); // 错误示范
```

上述错误的写法，编译器是不会提示警告或者错误的，并且在模拟器中也不会暴露出问题来。在发布自己的App前，一定记得要使用真机去测试。

总结

在支持64-bit过程中，应该按照Apple文档中提供的7个步骤完整检查项目工程。如果工程中涉及到大量的C或者C++代码，在支持64-bit中要更加谨慎。

写完这篇笔记后，我觉得需要重温一下C的基础知识。XD，顺便祈祷项目中的第三方库赶紧更新支持64-bit，阿弥陀佛。

ps: 找出不支持arm64的静态库 find . -name *.a -exec lipo -info "{}" \;

原文链接：http://chun.tips/blog/2014/10/21/iosgong-cheng-ru-he-zhi-chi-64-bit/

大规模网站架构的缓存机制和几何分形学

作者：徐汉彬

【导读】徐汉彬曾在阿里巴巴和腾讯从事4年多的技术研发工作，负责过日请求量过亿的Web系统升级与重构，目前在小满科技创业，从事SaaS服务技术建设。

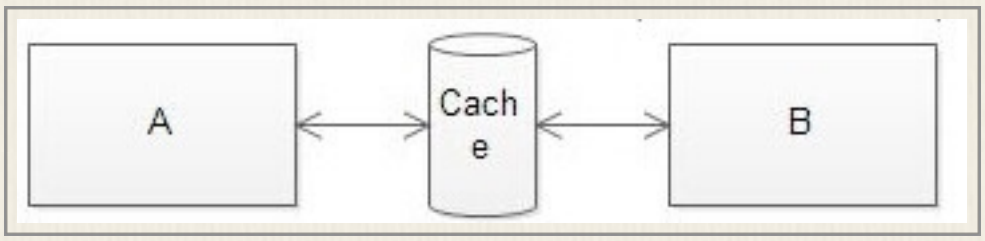
在过去的工作中，徐汉彬从事各类缓存建设和优化，遇到问题无数，从各种各样的问题中，逐渐总结出它们之间的“共性”，而这个“共性”又优雅地遵循“几何分形 学”。从几何分形的角度去看待缓存机制，能够更容易和更清晰地表述出它的深层原理和部署思想。帮助技术人员去解决在缓存上遇到的技术问题。

缓存机制和几何分形学

缓存机制在我们的实际研发工作中，被极其广泛地应用，通过这些缓存机制来提升系统交互的效率。简单的总结来说，就是在两个环节或者系统之间，会引入一个cache/buffer做为提升整体效率的角色。

而有趣的是，这种缓存机制令人惊奇并且优美的遵循着“几何分形”的规律，也就是几何分形学中的“自相似性”：从整体上看遵循某种组成规律或者特性，同时从每 一个局部看，仍然遵循某种组成的规律或者特性。我们的这些系统，从整体上看遵循了缓存机制，每一个组成的局部也遵循缓存机制。

等同类比的一个概念，我们常常说的“空间换时间”，牺牲一部分空间代价，来换取整体效率的提升。



例如A和B两者之间的数据交换，为了提升整体的效率，引入角色C，而C被用于当做热点数据的存储，或者是某种中间处理的机制。

我们先从web前端层面开始，看看有哪些比较关键的缓存机制？它们又是怎样协调工作的呢？

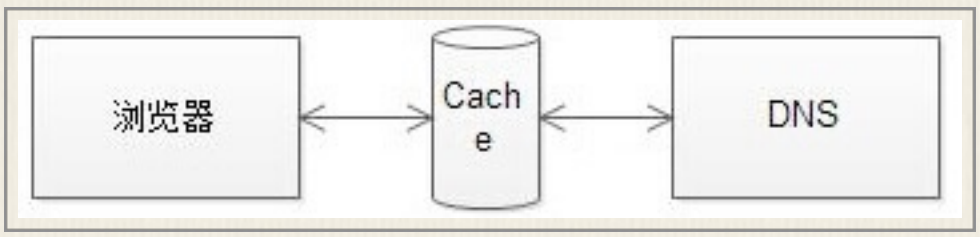
一、前端Cache机制

1. 域名转为IP地址（域名服务器DNS缓存）

我们知道域名其实只是一个别名，真实的服务器请求地址，实际上是一个IP地址。获得IP地址的方式，就是查询DNS映射表。虽然这是一个非常简单的查询， 但如果每次用户访问一个url都去查询DNS一次，未免显得太频繁，会产生一个可怕的访问量级。DNS服务器会告诉你，你别老是经常过来，万一我挂了，我们就无法愉快地玩耍了。

各个浏览器的缓存时间，会有一定的差别。例如，在chrome浏览器中查看dns的缓存时间的方式是：chrome://net-internals/#dns。

浏览器一般会在本地会建立一个DNS缓存，在一段比较长的时间里，都是使用本地的缓存映射。例如，在Win7系统的cmd里，可以通过“ipconfig /flushdns ”的方式来立刻刷新本地DNS。

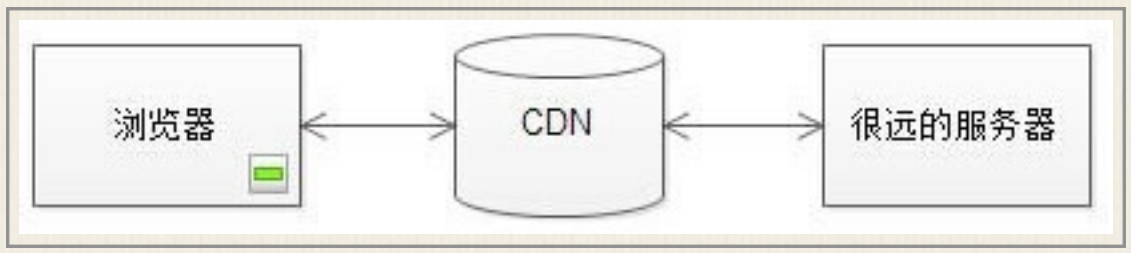


优点：域名映射为IP非常快。

成本：消耗一定的浏览器空间来存储映射关系

2. 访问服务器，获取静态内容（地理位置分布式服务CDN）

可能有人会觉得，这个CDN不是缓存。其实，CDN的原理就是将离你很远的东西，放在离你很近的地方，通过这种方式提高用户的访问速度。从这个角 度，它也可以理解为牺牲空间成本换取了时间，本质上也是一种特殊的中间cache。腾讯、阿里等这些大的一线互联网公司一般倾向于自己建立CDN系统，中 小型企业也经常使用第三方的CDN服务。



优点：解决用户离服务器太远的时候，网络路由中跳来跳去的严重耗时。

成本：全国各地部署多套静态存储服务，管理成本比较高，发布新文件的时候，需要等待全国节点的更新等。

3. 浏览器本地缓存（无网络交互类型）

在前端优化原则中，其中一条就是尽量消灭请求，以达到降低服务器压力和提升用户体验的效果。静态文件，例如Js、html、css、图片等内容，很多内容可以1次请求，然后未来就直接访问本地，不再请求web服务器。

常用的实现方法是通过Http协议头中的expire和max-age来控制，这两者的使用方法和区别，我这里就不赘叙了。还有一种HTML5中很热的方式，则是localStorage，尤其在移动端也被做为一个强大的缓存，甚至当做一种本地存储来广泛使用。

优点：减少网络传输，加快页面内容展示速度，提升用户体验。

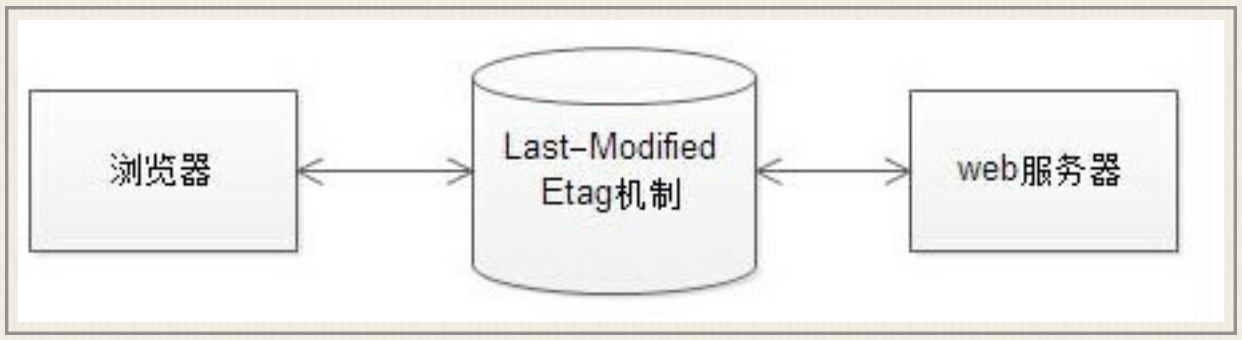
成本：占用客户端的部分内存和磁盘，影响实时性。

4. 浏览器和web服务协议缓存（有网络交互类型）

浏览器的本地缓存是存在过期时间的，一旦过期，就必须重新向服务器请求。这个时候，会有两种情形：

- 服务器的文件或者内容没有更新，可以继续使用浏览器本地缓存。
- 服务器的文件或者内容已经更新，需要重新请求，通过网络传输新的文件或者内容。

这里的协商方式也可以通过Http协议来控制，Last-Modified和Etag，这个时候请求服务器，如果是内容没有发生变更的情况，服务器会返回 304 Not Modified。这样的话，就不需要每次访问服务器都通过网络传输一个比较大的文件或者数据包，只要简单的http应答就可以达到相同的请求文件效果。



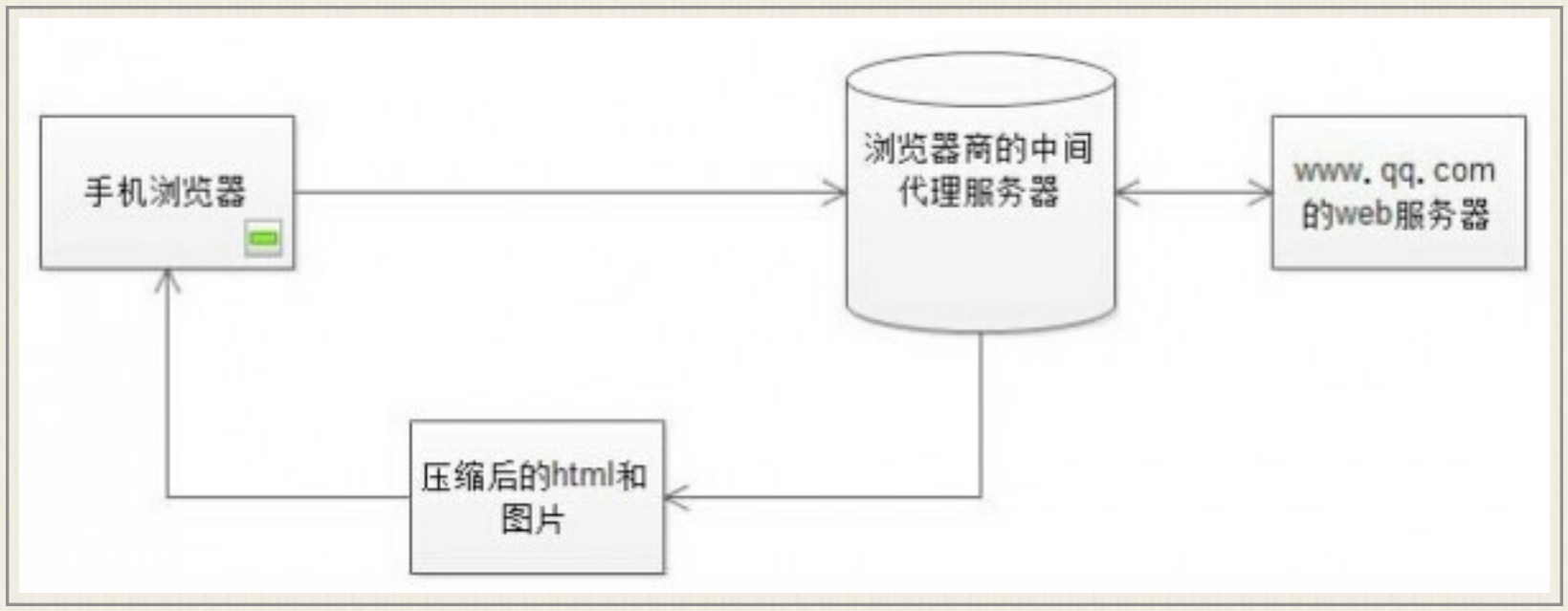
下图中的例子，是腾讯的自建CDN（imgcache.gtime.cn）：

优点：减少频繁的网络大数据包传输，节约带宽，提升用户体验。

成本：增加了服务器处理的步骤，消耗更多的CPU资源。

5. 浏览器中间代理

上面的几种cache机制，实际上都是非常常见。但是，在移动互联网时代，流量昂贵是很多用户心中深深的痛。于是，又出现了一种新型的中间cache， 也就是在浏览器和web服务器再架设一个中间代理。这个代理服务器会帮助手机浏览器去请求web页面，然后将web页面进行处理和压缩（例如压缩文件和图片），使页面变小，然后再传输给手机端的浏览器。



部分手机浏览器（例如Chrome）号称可以节省流量，提升访问速度，实际上就是上述做法。但是，也分为两种情况：

- 用户的网络和手机配置都比较差，因为页面被压缩变小，加载和传输速度变快，并且节约了流量。
- 用户的网络和手机配置都比较好，本身直连速度已经很快了，反而因为设置了中间代理，加载速度变慢，也可节约流量。

下图是chrome手机浏览器中，开启和不开启中间代理的对比图：

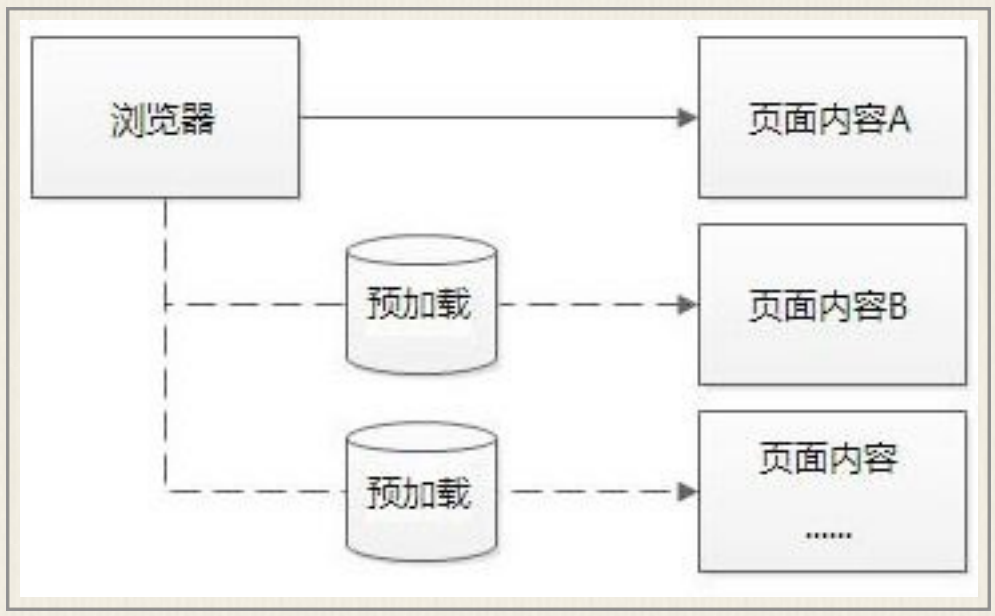
优点：节约用户流量，大部分情况下提升了加载速度。

成本：需要架设中间代理服务器，对各种文件进行压缩，有比较高的服务器维护成本。

6. 预加载缓存机制

这种加载方式主要流行在移动端，为了解决手机网速慢和浏览器加载性能问题，浏览器会判断页面的关联内容，进行“预加载”。 也就是说，在用户浏览A页面的时候，就提前下载并且加载B页面的内容。给用户的体验就是，B页面一瞬间就出现了，中间没有任何延迟的感觉，从而带来更好的 极佳的用户体验。

这种实现机制，往往由浏览器来实现，当然，手机页面本身，也可以通过JS来自身实现。而这种机制也存在一些问题，浏览器需要预判用户的浏览行为，在一些场景下，这个预判算法本身不一定准确，如果不准确则带来一定的流量、内存和系统资源的浪费。



优点：给用户带来极佳的页面展示体验。

缺点：预判实现比较复杂，占据一定的内存和手机系统资源，可能产生流量和资源浪费。

前端的cache当然不仅仅如此简单，如果细致到每一个小环节和组成部分，我们会发现实际上是无处不在的，例如浏览器的渲染行为、网络网卡的传输环节，小环节和小环节之间也有无数这种类型的cache角色。

这个就如同几何分形学中的自相似性：从整体上看符合某种组成规律或者特性，同时，从局部看，仍然符合某种组成的规律或者特性。

几何分形的现象在我们生活中，也是非常常见的，例如：

人体中的几何分形例子，例如：人体有1个头部+4肢，局部上看人的手指也是1个手指头+4个手指；人体无论整体或者局部，都大致遵循黄金分割点0.618的比例来生长（五官按照这个比例越多，越好看）。

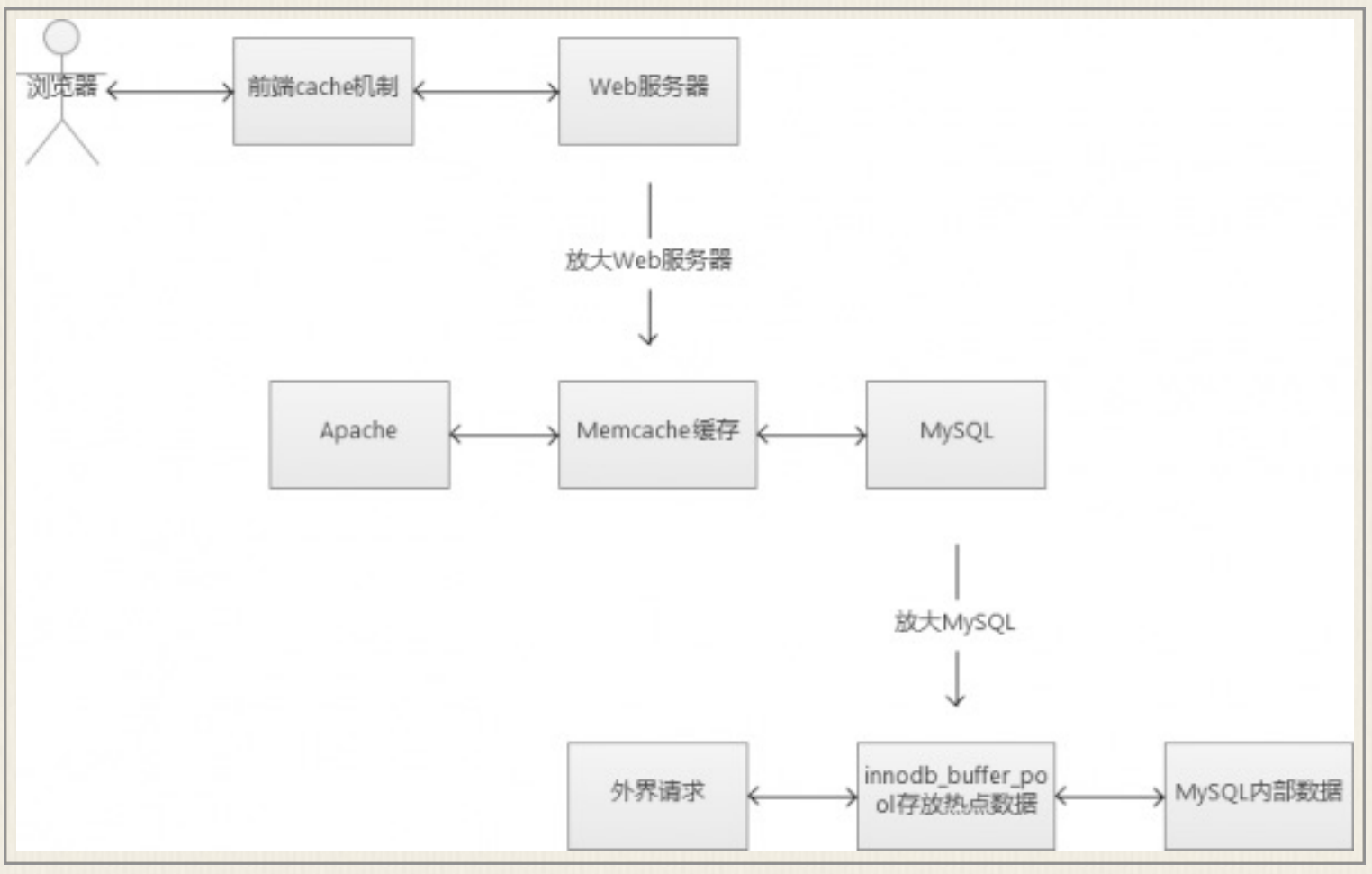
例如下图中的叶子，每个局部都和主干组成结构相似。

二、Web系统和几何分形学

1. Web系统中的缓存机制

看完上面的前端cache，我们会感觉到缓存机制在前端中的确无处不在，那么它在其他地方和环节，是否也无处不在?

可以看看这张图：



实际上，每一个环节本身是可以又再次被放大的，放大以后，我们又看见了更多缓存机制的“特性”存在。从一个整体来看，符合该规律，从组成部分来看，仍然符合该规律。

每一个组成缓存机制的“成员”的内部，又存在着更多的缓存机制。

Apache内部的一些“缓存机制”：

- url映射缓存mod_cache（有mode_disk_cache和mod_mem_cache，后者官方已不推荐）
- 缓存热点文件打开描述符mod_file_cache（对于静态文件的情况，减少打开文件中open行为的耗时）
- 启动的时候，通过prefork模式设置的Start-Servers服务进程池，牺牲内存空间。

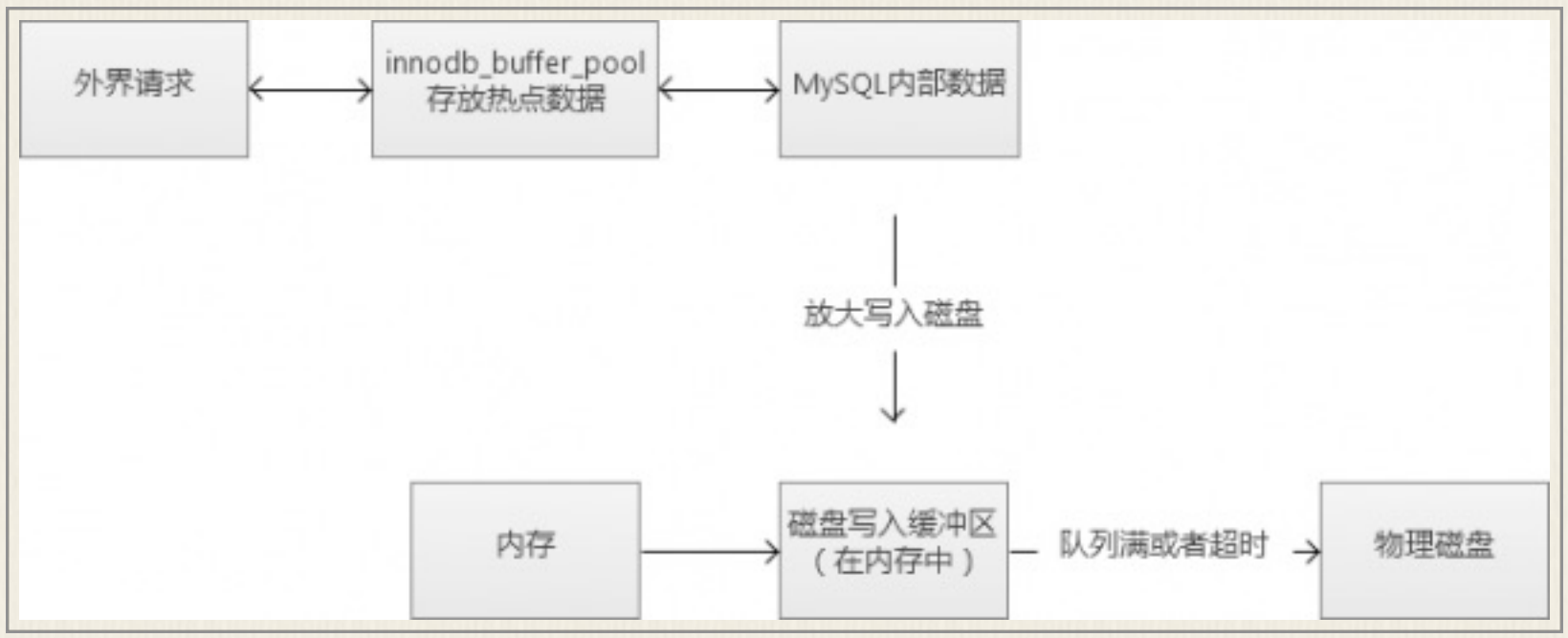
MySQL内的一些“缓存机制”：

- 数据库的索引，牺牲磁盘空间（组合索引等会占据很大的磁盘空间）
- innodb_buffer_pool_size，热点数据的缓存，牺牲内存空间
- innod-b_flush_method写入磁盘的机制，可以配置成缓冲写入的方式
- query_cache_size查询缓存，牺牲内存空间
- thread_cache_size数据库连接池的缓存个数，牺牲内存空间

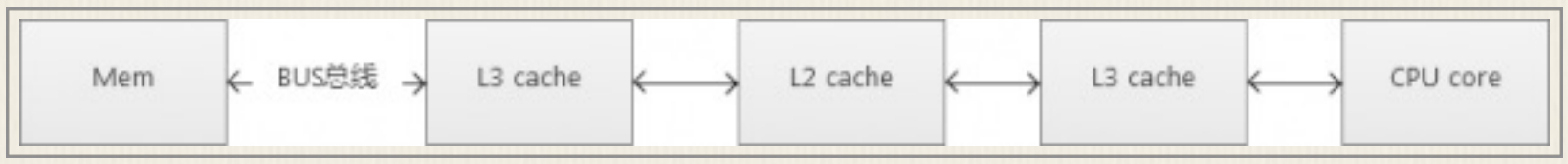
2. 接近硬件层面的“空间换时间”

那我们再来看更细小的一个环节，计算机写的操作。我们会发现，在内存和物理磁盘之间，还有一个磁盘缓冲区（页高速缓存）的存在，这个是内存和磁盘之间的“缓存”。当然，读取的操作也是同理。

下图是“放大”MySQL中的写入磁盘：



实际上，更进一步看，CPU和内存之间也存在缓存机制（常用指令会存在放在寄存器中，因为CPU访问寄存器会远快于访问内存，中间为了缓冲它们之间差距，设置了多级高速缓存）。



例如下图是Intel i7 920的各级缓存大小：

CPU内核	核心代号	Bloomfield
	CPU架构	Nehalem
	核心数量	四核心
	线程数	八线程
	制作工艺	45纳米
	热设计功耗(TDP)	130W
	内核电压	0.8-1.375V
	晶体管数量	731百万
	核心面积	263平方毫米
CPU缓存	一级缓存	128KB
	二级缓存	1MB
	三级缓存	8MB

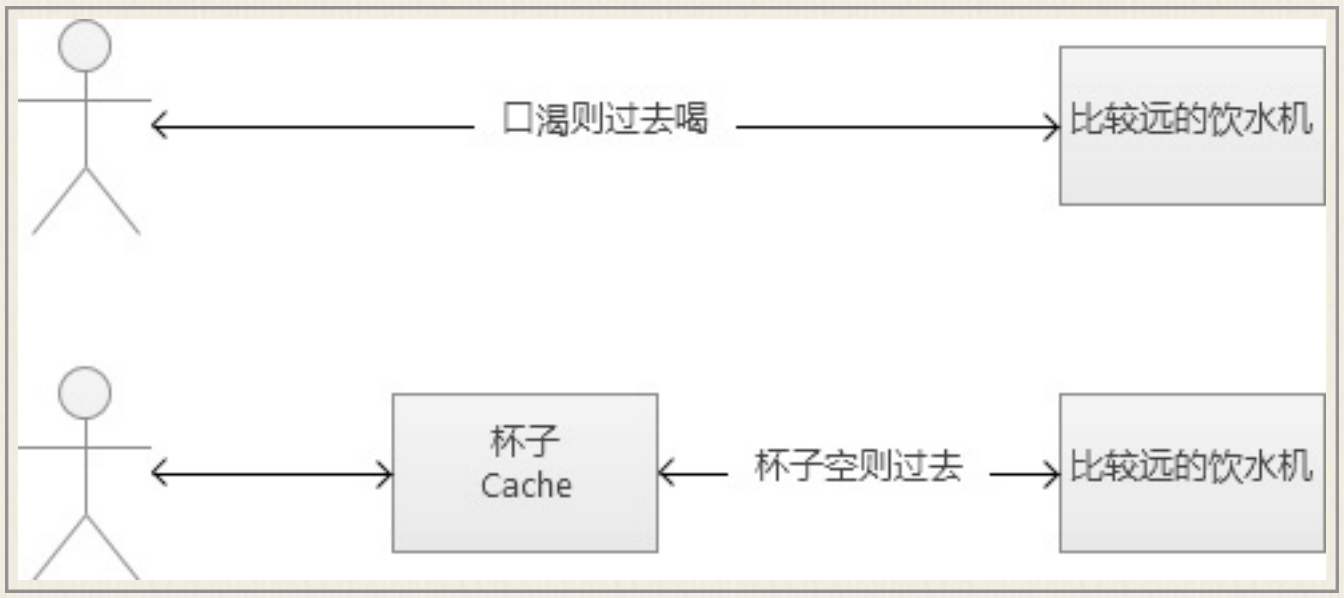
这个时候，我们可以看出来，计算机系统从大的系统层面看，是遵循“缓存机制”的规律的，同时，在每个局部成员的层面，同样遵循该规律。

3. 现实世界中的“缓存机制”

我们现在喝水通常使用的是杯子，杯子实际上也扮演着一个特殊的Cache角色。举个例子：一个人离饮水机比较远，他渴了，他有如下两种“喝水”的方式：

- 不用杯子，每次渴了直接去饮水机喝（这个比较霸气侧漏，不要在意细节）。结果：频繁跑动，耗费体力。
- 使用杯子，渴了先喝杯子（Cache）上的水，如果杯子没有，带上杯子去装水，再喝。结果：比较少跑动，节省体力。

这样看不直观，简化为一个流程图如下：



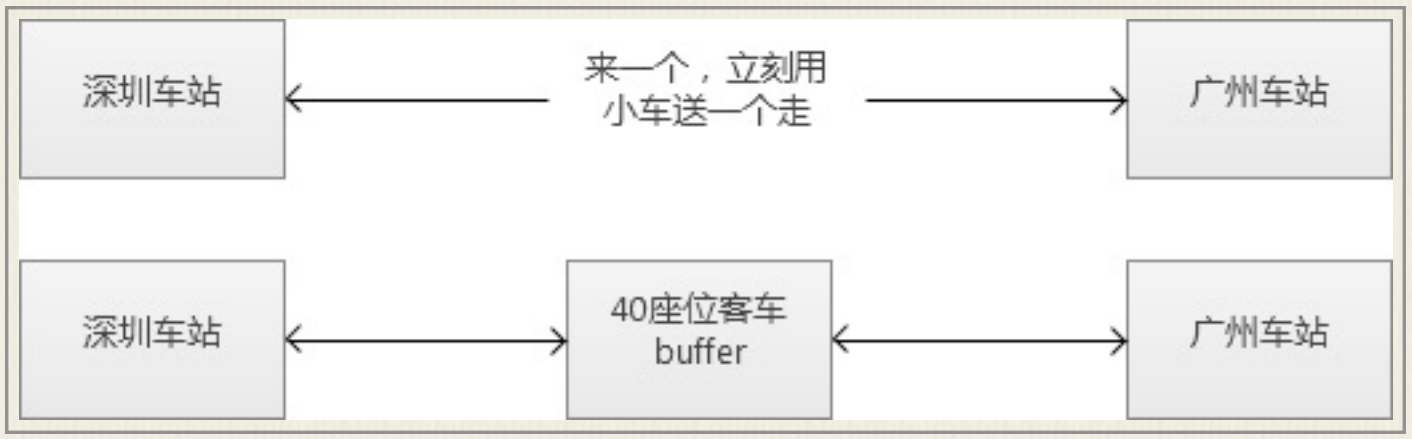
这虽然是个人尽皆知的道理，但是，这个方法本身是“进化”出来的。百万年前的原始人类和其他大自然的动物一样的，喝水遵循了第一种方式，只是随着人类的发展，“进化”出第二种喝水的方式。

这里也存在一个缓存机制，就是用杯子的空间获取喝水效率的时间。

还有一个更为典型的例子，就是坐车/运输，假设我们从深圳去广州，我们会去坐客运车。而客运车（假设上面有40个 座位）实际上相当于一个40个座位的“队列”。遵循着网络传输的相同的规律“队列满或者超时则发送”。客车本身的40个位置，就像一个“发送缓冲区”。使 用和不使用这个大的缓冲区，客车也可以有两者运作方式：

1. 车站发现来一个人，用只能容纳一个人的小车，不等待直接送一个人去广州。

2. 车站发现来一个人，先放进客车buffer中，等待人满或者达到班车约定时间（队列超时）再出发。



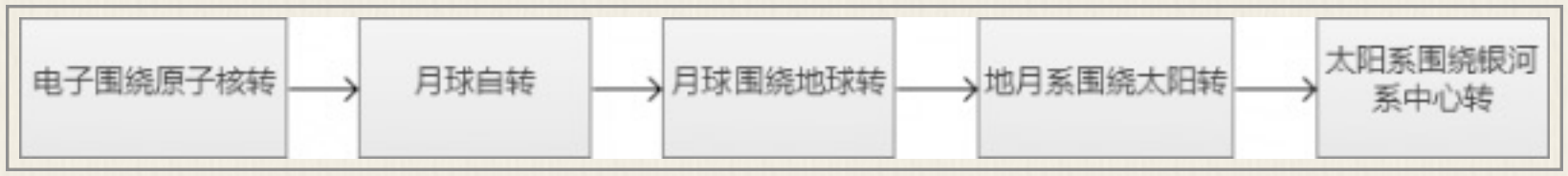
显而易见，第一种是太浪费资源了。

除此之外，还有很多各种各样的例子，如江河上的大坝、我们桌面上的一些东西（它们占据宝贵的桌面空间）、我们公司附近小店里的商品、离我们近的东西等等。

看到这里，很多人会渐渐发觉，计算机的一些原理，竟然在现实世界里有无处不在的“映射和影子”。

几何分形学是个非常有趣的东西，某些规律，实际上还贯穿在整个宏观和微观世界中。

例如“绕转”的现象：



4. 现实世界和计算机“缓存机制”原理的关系，为什么遵循“几何分形”?

实际上，计算机的原理来源于数学，而数学是日常生活现象和规律的高度抽象，源于生活，高于生活。

同时，不仅仅“缓存机制”，还有多线程等原理，也能找到这种遵循“几何分形学”的样子。

“缓存机制”简单总结可以说是“空间换时间”，我在大学第一天看见这句话的时候，我觉得自己一看见就懂了。随着对技术理解的加深和工作经验的积累，我渐渐明白当时我理解得太浅了。

关于作者：徐汉彬，曾经在阿里巴巴和腾讯有过4年的技术研发工作经历，目前在小满科技（创业）。

原文链接：http://www.csdn.net/article/2014-10-21/2822218